LINKING TCAD AND EDA THROUGH PATTERN MATCHING

by

Frank Edward Gennari

Memorandum No. UCB/ERL M04/31

3 August 2004

COVER

LINKING TCAD AND EDA THROUGH PATTERN MATCHING

by

Frank Edward Gennari

Memorandum No. UCB/ERL M04/31

3 August 2004

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Linking TCAD and EDA through Pattern Matching

by

Frank Edward Gennari

B.S. (Carnegie Mellon University) 2000
M.S. (University of California, Berkeley) 2001

A dissertation submitted in partial satisfaction of the requirements for the degree of

Doctor of Philosophy

in

Engineering-Electrical Engineering and Computer Sciences

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Andrew R. Neureuther, Chair
Professor Alberto Sangiovanni-Vincentelli
Professor Kameshwar Poolla

Fall 2004

The dissertation of Frank Edward Gennari is approved:

Chair	Date
	Date
	Date

University of California, Berkeley
Fall 2004

Linking TCAD and EDA through Pattern Matching

Abstract

Linking TCAD and EDA through Pattern Matching

by

Frank Edward Gennari

Doctor of Philosophy in Engineering-Electrical Engineering and Computer Sciences

University of California, Berkeley

Professor Andrew R. Neureuther, Chair

As the critical dimension in optical lithography shrinks to 90nm and below, determining where the layout is most affected by non-ideal process conditions is increasingly important. In many cases, combinations of local layout geometries that produce or are sensitive to residual effects can be found by locating theoretically problematic configurations of shapes. This dissertation explores the architectural, physical, and algorithmic feasibility of a prototype pattern matching approach as a novel Technology Computer Aided Design (TCAD) tool for linking to Electronic Design Automation (EDA).

The pattern matcher software architecture was created as a standalone Design for Manufacturability (DFM) tool that fits easily into the design flow and can be applied to many areas of lithography and integrated circuit processing. The development was motivated by the need to relate residual lens aberration effects back to the layout design. For this application, the pattern generator first reads a set of Zernike polynomials and takes the inverse Fourier transform (IFT) of the aberrated pupil function in order to generate the pattern bitmap. The pattern matcher then loads the pattern, a user input parameter file, and a multilayer mask layout in CIF or GDSII format. The match factor is computed along each edge and at each corner of interest, and a resulting sorted table of highest match factors for each pattern is output. The matching geometry is extracted for more rigorous process simulators such as SPLAT. The system also supports an interactive graphical display of the layout with pattern images drawn over the match locations.

In assessing the physical feasibility of this approach, it has been determined that aberrations produce half the line edge shift as optical proximity effects, and the pattern matcher has been verified to accurately predict electric field change through comparison with SPLAT simulations. Other applications of pattern matching include analyzing effects of misalignment, defects, reflective notching, laser-assisted thermal processing, Chemical-Mechanical Polishing (CMP) dishing, and flare, some of which involve processing multiple layers and performing Boolean layer operations.

The key contribution to this thesis is the collection of data structures and algorithms that implement pattern matching. The algorithm requirements of searching for images in layouts differ significantly from image correlation, video compression, and geometric matching methods due to the enormous search space, large groups of identical pixel values, complex pixel and layer weights, inexact matching, and filtering methods used. The novel techniques developed for pattern matching include spatial sorting and subdivision of the layout, pre-integration of the pattern, triangulation of polygons, layer Boolean operations, and pre-filtering of match locations. The most efficient matching algorithm uses rectangle and triangle primitives and can efficiently process an entire chip in less than an hour on a standard desktop computer with near perfect scaling on parallel processor machines. This runtime is two orders of magnitude faster than Optical Proximity Correction (OPC).

Professor Andrew R. Neureuther, Committee Chairman

Acknowledgements

This research was funded by the Semiconductor Research Corporation (SRC) contract 01-MC-460 and DARPA grant MDA972-01-1-0021. I would like to thank Professor Andrew Neureuther and his research group for their help in writing the software and explaining various lithography concepts. I would like to thank Garth Robins, Michael Lam, and Kostas Adam for their help with the pattern generation and theory, and Scott Hafeman for his assistance with SPLAT and flare. I would also like to thank Matt Moskewicz for his assistance with the C++ programming, and Shariq Rizvi, and Guille Diez-Canas for their help with the parallel pattern matcher code. I would like to thank Jonathan Hsu, Sachan Madahar, and the other undergraduate system administrators/web developers for creating the web interfaces to the pattern matcher.

Table of Contents

Index of Figures

Index of Acronyms

ACLV	Across-Chip Linewidth Variation
BARC	Bottom Anti-Reflective Coating
BMP	BitMaP
CAD	Computer Aided Design
CD	Critical Dimension
CGI	Common Gateway Interface
CIF	Caltech Intermediate Format
CITRIS	Center for Information Technology Research in the Interest of Society
CMP	Chemical-Mechanical Polishing
DFM	Design for Manufacturability
DFII	Design Framework II
DRC	Design Rule Check
EDA	Electronic Design Automation
ESF	Edge Skip Factor
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
GDSII	Graphical Data Stream II
GLUT	Graphics Library UTilities (for OpenGL)
GUI	Graphical User Interface
IFFT	Inverse Fast Fourier Transform
IFT	Inverse Fourier Transform
JPEG	Joint Photographic Experts Group
LAP	Laser-Assisted Processing
LAVA	Lithography Analysis using Virtual Access
LTP	Laser Thermal Processing
LOD	Level of Detail
MF	Match Factor
MLTP	Maximal Lateral Test Pattern
MPI	Message Passing Interface
NA	Numerical Aperture
NERSC	National Energy Research Scientific Computing Center
NGC	Normalized Grayscale Correlation
OASIS	Open Artwork System Interchange Standard
OPC	Optical Proximity Correction
OpenGL	Open source Graphics Library
OPD	Optical Path Difference
OTF	Optical Transfer Function
PC	Personal Computer
PEM	Phase Edge Mask
PPC	Process Proximity Correction
PSF	Point Spread Function
PSM	Phase Shift Mask

Pthreads	POSIX threads
RET	Resolution Enhancement Technique/Technology
RMS	Root Mean Square
SEM	Scanning Electron Microscope/Micrograph
SEMI	Semiconductor Equipment and Materials International
SIMPL-2	SIMulated Profiles from the Layout version 2
SPIE	Society of Photo-Optical Instrumentation Engineers - The International Society for Optical Engineering
SPLAT	Simulation of Profile and Lithography Analysis Topography
STL	Standard Template Library
TCAD	Technology Computer Aided Design

1 Introduction

1.1 Orientation

1.1.1 Motivation

In order to enable the next generation of integrated circuit technology, it is important to find an efficient way of linking residual processing effects back to the circuit design stage. In this way, information can flow back to designers so that problems can be corrected early in the design. This is the fundamental concept of Design for Manufacturability (DFM). Current Electronic Design Automation (EDA) tool features such as Optical Proximity Correction (OPC) typically only deal with first order processing effects and neglect the impact of effects such as lens aberrations on the printed image. As minimum feature sizes continue to shrink to the 90nm node and below, it is becoming increasingly important to take these processing effects into account within the EDA tool environment of the designers. The pattern matching system presented in this dissertation is a valuable way of linking Technology Computer Aided Design (TCAD) with EDA.

1.1.2 Problem

Residual processing effects such as lens aberrations, flare, and reflective notching cause unwanted effects such as line edge shift and critical dimension (CD) variation. EDA tools exist that correct for the optical proximity effects and other well-modeled and predictable problems, but there are relatively few software tools that take into account other difficult to predict, not well understood, or secondary non-idealities in integrated circuit manufacturing. Predicting the results of these effects requires time-consuming simulations that often cannot be run full chip and otherwise require tedious user input to limit the simulation area. Because of their complexity, standard design rules cannot locate areas sensitive to these effects, especially the statistical problems. Furthermore, many of these process correction tools occur late in the design stage after a significant amount of time and effort have been spent on a problematic design, requiring expensive mask rewrites. In addition, the number of circuit features that need to be taken into account when predicting the results of these processing effects is continuously growing as the layout scales faster than the area of influence. The adoption of phase-shifting masks not only contributes to the complexity of processing effects such as lens aberrations, but it also makes the effects more severe.

These lithographic and processing effects cannot be neglected in the design of modern chip layouts in the sub-100nm, low k1 regime where feature widths are less than a quarter the wavelength of the light used to print them. Ignoring the effects of aberrations and other residual processing effects may result in reduced reliability and yield loss in the fabricated design or an increased number of mask rewrites. B. Grenon [1] reports that masks must often be written three times and, in some cases, as many as seven times. It is becoming more and more critical that these effects be taken into account when designing the circuit layout. Conservative over-specification of the design through tight design rules is not the answer to this problem.

1.1.3 Approach

If the geometries that are most susceptible to residual processing effects can be determined, then a search can be made to locate all areas in an integrated circuit layout design that are similar to these sensitive geometries. This search can be performed for a very general class of physical effects by pattern matching a bitmap representation of the theoretically most sensitive pattern on the layout. All matches found are potential problem areas that must be examined in more detail by the designer or process engineer to determine if action needs to be taken. These problematic geometries can then be modified to reduce the layout's overall susceptibility to process-induced manufacturability issues. This software system allows designers to utilize processing information developed by technologists so as to produce lithography-friendly designs. Pattern matching can thus be used to determine when a design is ready for tape-out. Alternatively, these areas predicted to be most affected by aberrations and other residual processing effects could be recorded and later used to reduce the number of locations that must be inspected in the printed layout.

This pattern-matching procedure can potentially reduce the effects of imperfect optics and other integrated circuit manufacturing equipment on the printing of the image from the mask to the wafer, improving the yield of the design. Patterns can even be generated from the quantitative description of the manufacturing equipment that will be used. This allows a designer to tailor the design to be robust to the limitations of the particular machines that will be used to manufacture it. Non-ideal processing effects can be compensated for in a way similar to OPC.

The pattern matching system is fast enough to be run on a full chip layout and exists entirely in software thus does not require printing of test wafers or other expensive procedures. Pattern matching is therefore a fast and low cost method of improving the robustness of integrated circuit layouts early in the design cycle.

1.2 Contributions of this Dissertation

The goal of this research was to create a complete system based on a sound theoretical background that provides an efficient way of locating layout geometries that are the most affected by physical effects involved in the manufacturing of semiconductors. The contributions of this dissertation can be broken down into three areas: architectural, physical, and algorithmic. The architectural contributions involve the creation of a pattern matching system that allows a user to analytically specify a pattern, import layout geometry in a standard format, process the layout, and receive a list of locations that meet the matching criteria. The physical aspects of this project can be broken down into the theoretical derivation of the patterns used in the matching, testing on various layouts, and verification of the match results against simulation. The algorithmic contributions include the bitmap, edge, rectangle, and triangle algorithms along with their data structures and supporting geometric algorithms. The following sections describe the pattern matching approach, the system architecture used to implement this approach, the algorithms used for efficient pattern matching, and the physical testing used to verify the accuracy and importance of the pattern matching software system.

1.2.1 Pattern Matching Approach

This research project began with the pattern matcher theory and the initial application of determining areas in a mask layout sensitive to lithographic lens aberrations. Finding a quantitative method of measuring and reducing the layout's sensitivity to lens aberrations at the design stage had not previously been done and was thus a motivation for this research.

The lens aberration application was the driving force behind the pattern matcher project and defined the initial goals and requirements of the system. Using the theory developed by Robins, Neureuther, and Adam [30] that the inverse Fourier Transform of Zernike polynomials over a circular pupil provides the aberrated point spread function (PSF), a library of patterns has been generated for representing various lens aberration terms. Extensions to the aberration theory have been applied to generate equations used to derive the electric field change, intensity change, and line edge shift from the calculated match factor.

The lens aberration pattern matching idea has been generalized to a Maximal Lateral Test Pattern (MLTP) representing the worst case geometries for a number of residual processing effects. MLTPs have been formulated and patterns generated for applications such as flare, reflective notching, laser-assisted processing, CMP dishing, defects, and misalignment.

1.2.2 Architecture

A software system has been written which implements efficient pattern matching algorithms in order to test the pattern matching theory. This system includes a program for generating aberration patterns from the inverse Fourier transform (IFT) of Zernike polynomials and flare patterns from the IFT of scattered light equations. It also includes a tool for loading CIF and GDSII layouts and either storing them hierarchically or converting them to other formats. The core pattern matcher executable then reads the layout and patterns and produces various text files and one or more images of the results. The system utilizes a custom high-performance OpenGL interactive graphical display system developed for viewing the pattern matching results. Layouts, patterns, and various simulation results can also be viewed with these display programs. Together, these components form a complete software system that directly reads a layout in standard industry formats and a set of pattern equations, producing both textual and graphical results of all locations in the layout that are sensitive to the processing effect described by the pattern equations. Figure 1 gives an overview of the system architecture as used for determining sensitivity to lens aberrations. A more thorough description of the software is given in Chapter 1.

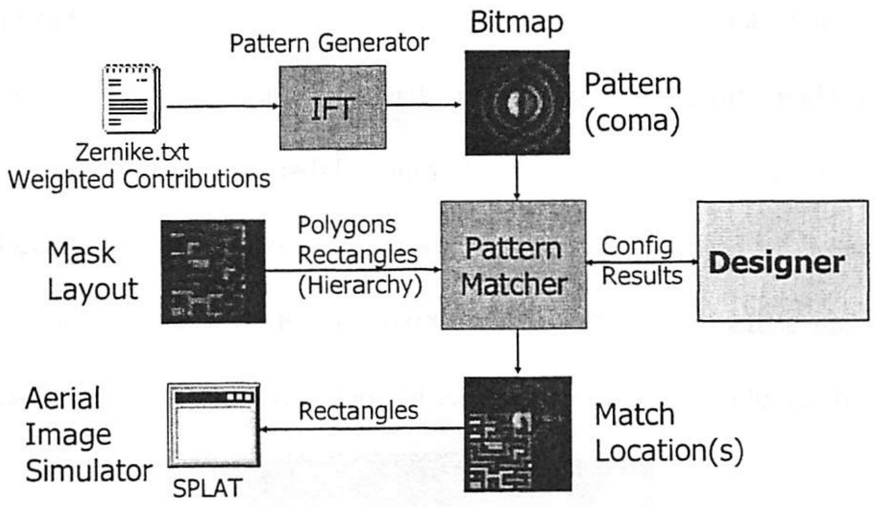


Figure 1: Pattern matcher system architecture for detecting areas sensitive to lens aberrations

1.2.3 Algorithms

One of the most important contributions to this research is the set of algorithms that has been implemented in the core pattern matcher executable. Four generations of algorithms have been developed, each much more efficient than the last: the bitmap algorithm, edge-intersection algorithm, rectangle algorithm, and rectangle/triangle algorithm. The algorithms involve pre-integrating the pattern matrices, splitting polygons into smaller primitives, spatially partitioning and sorting the layout geometry, computing Boolean operations on layers, and filtering match locations by various methods. The novel algorithms that have been developed for efficient pattern matching include not only the core pattern matching components but also the method of computing layer Boolean and other operations on rectangles by recursively spatially subdividing the layout into small areas that can be processed very quickly and stitched back together. The unique polygon splitting algorithm can divide an arbitrary polygon into a minimal set of simpler shapes through repeated horizontal and vertical subdivision into smaller polygons until the sub-polygons are simple rectangles and right triangles. Another important component of this research is the compressed hierarchical database structure that allows for efficient query of pattern matching results, and the related interactive layout graphical display engine.

These algorithms combined allow the program to efficiently process a full chip layout in only minutes on a standard desktop PC or workstation. Abacus.gds[1], one example of a full chip GDSII layout that the pattern matcher has been run on, is shown in Figure 2.

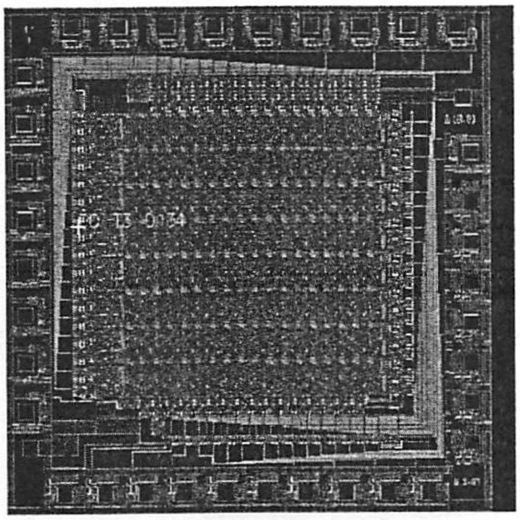


Figure 2: Abacus GDSII layout, 3mm by 3mm, 2μm technology, 14 layers, 5 levels of hierarchy[1]

1. Example from Dolphin Integration, http://www.dolphin.fr/medal/socgds/socgds_free_overview.html

1.2.4 Physical Testing

A significant contribution to this dissertation involves the physical verification of the aberration pattern matcher theory. Experimental simulations have been performed on a number of test layouts that show the effect of lens aberrations on feature edge positions is significant, resulting in as much as half the line edge shift as optical proximity effects even for high-quality optical lenses. SPLAT aerial image simulations have been used to analyze the effects of geometry on line end and line edge shift due to even and odd lens aberrations such as coma.

In addition to verifying the importance of aberrations, the accuracy of the pattern matcher in predicting the geometries sensitive to aberrations has been proven through simulation. These experiments involved running the pattern matcher on a large number of hand drawn, script generated, and real industry layouts. The pattern matcher was found to predict electrical field changes that agreed well with SPLAT simulations, and these electric field changes can be extended to the computation of intensity changes and line edge shifts.

Some other pattern matching applications have also been experimentally verified to some extent. The effects of flare at various magnitudes and coherence lengths were analyzed in a number of situations. The pattern matcher was found to locate areas in a layout sensitive to flare as well as optical proximity effects. In addition, pattern matching runs have been performed on hand-crafted test layouts for applications such as laser-assisted processing, reflective notching, and CMP dishing.

1.3 Dissertation Organization

This dissertation begins with background information and previous work in the relevant areas of lithography in Chapter 2, including lens aberrations and Optical Proximity Correction. Chapter 3 discusses the theoretical background of pattern matching for the application of lens aberrations, the meaning of the match factor, and the definition of a Maximal Lateral Test Pattern. Chapter 4 describes the pattern matcher software system and how each of the components contributes to a complete EDA tool. Chapter 5 explains how the pattern matching software has evolved from a simple tool that was integrated into the Cadence Design Framework II to a standalone graphical system. Chapter 6 compares in detail the various pattern matching algorithms that were examined and the strengths, weaknesses, and performance of each. Chapter 7 describes the details of the software data structures and algorithms, including some of the novel features of the system. Chapter 8 presents results that show the importance of lens aberrations, verify the accuracy of the match factor prediction, and give performance numbers on real full-chip layouts. Chapter 9 extends the pattern matching idea to other areas of lithography and wafer processing. Finally, concluding remarks are given in Chapter 10.

2 Background

This dissertation involves a broad range of topics from several engineering fields, including lithographic issues such as lens aberrations and OPC, computational geometry, computer science algorithms, CAD/EDA tools, and computer graphics. Some of these areas and additional background material are introduced in the following sections.

2.1 Lithography

Most of the pattern matching applications discussed in this dissertation are in the area of lithography. The pattern matcher system was designed to locate areas in a layout that are sensitive to problematic effects somewhere in the process of turning a set of polygons stored on a designer's hard drive into a printed image on a wafer. These circuits contain polygons that will eventually be printed as tiny metal and polysilicon lines on the order of 100nm or less in width, or around one thousandth the width of a human hair. Recent technology nodes involve printing line widths smaller than the wavelength of the light used, which requires a significant number of Resolution Enhancement Techniques (RETs) [2, 3, 4]. The polygons representing an integrated circuit layout are first written to a mask, which is then used to print a large number of wafers through an extremely complex lithographic process. The wafers are then cut into individual chips that are packaged into integrated circuits such as microprocessors and memory. The lithographic process involves a stepper or scanner projecting light through the openings on the mask, through a complex system of lenses that reduces the size of the image, and onto the wafer, where the light exposes the photo-sensitive resist and creates images approximating the original mask polygons.

This process is by no means easy or exact. The lithography process is a limiting factor in determining the critical dimension of the technology, and thus is continuously pushed to its limits to further reduce feature sizes. As these feature sizes shrink, the effects of polygons on their neighbors become increasingly strong. The results of an imperfect optical path and of adjacent feature interactions are difficult to predict at the full chip level, but are extremely important in creating a manufacturable design with good yield. Thus it is critical to determine which geometries are prone to adverse lithographic process effects early in the design cycle, before the expensive masks are even written.

2.2 Optical Proximity Effects

Lithographic projection printing systems employ a series of lenses to send light through the mask and produce a de-magnified image in the resist on the wafer. Light entering the openings in the mask produces plane waves that are the Fourier transform of the mask geometry. The lens then low-pass filters and inverse Fourier transforms (IFTs) the transmitted plane waves to reproduce an image in the resist resembling the original mask openings. Several problems arise from the small size of modern integrated circuit device features and the finite size and inherent limitations of the imaging systems. First, since the lens system is diffraction limited, the high spatial frequency components required to reproduce the sharp edges in polygon features fall outside the lens. Secondly, stray light entering the opening from one shape may find its way into another shape in close proximity, leading to a complex interaction of the electric fields of adjacent polygons. Thus the final shapes will have rounded corners and may bulge towards adjacent shapes, possibly shorting together (bridging) and rendering the chip defective if the situation is severe.

Optical Proximity Correction (OPC) [5, 6, 7] is a step in the manufacturing process that semiconductor manufactures employ to improve the manufacturability of high-performance integrated circuit designs such as microprocessors. OPC is the process of modifying the polygons that are drawn by the chip designers in order to compensate for the limitations of the lithographic process assuming a perfect but finite lens. Given the shapes desired on the wafer, such as exact line widths and sharp corners, the mask is modified to improve the reproduction of the critical geometry [8, 9]. This is done by dividing polygon edges into small segments and moving the segments around, and by adding additional small polygons to strategic locations in the layout. The addition of OPC serifs and scatter bars to a layout is shown in Figure 3 [10]. The addition of OPC features to the mask layout allows for tighter design rules and significantly improves process reliability and yield. However, OPC also increases the data size and may destroy the hierarchy.

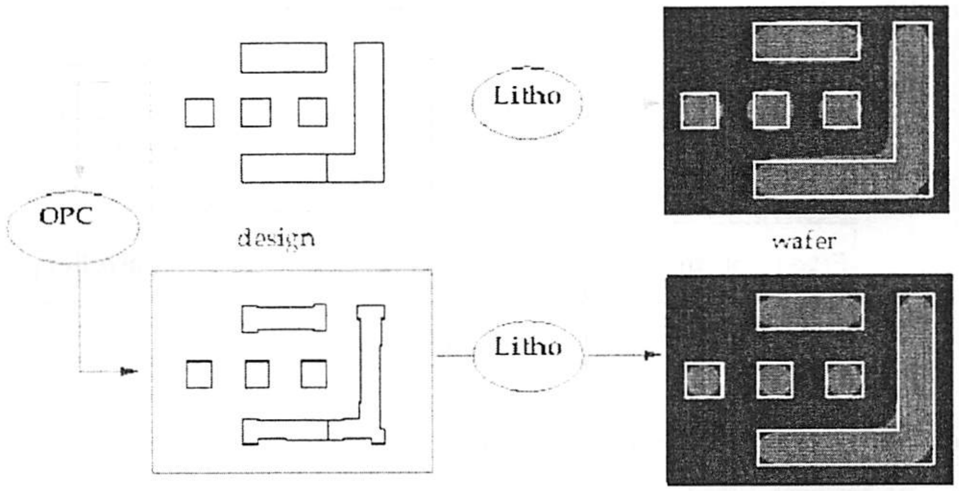


Figure 3: Addition of OPC shapes to layout for improved printed image (image taken from [10])

OPC is often run on the entire chip at once and iterated until convergence. There are many different types of OPC algorithms, the two main classifications being rule-based and model-based. Each involves subdividing polygons into smaller shapes or edge segments, known as fragmentation, and moving or adding shapes to other positions. Rule-based OPC is simpler in that various geometries are treated by different rules from a rule library. Model-based OPC, as demonstrated in Figure 4, is more complex and involves performing fast optical image simulations, which may be accomplished by computing a weighted sum of pre-simulated "OPC kernels" for simple edges and corners that are stored in a library. The proximity effects due to adjacent shapes is additive in electric field, but since the actual printed feature edges are related to light intensity $I = E \cdot E^*$, the nonlinear addition of terms requires multiple kernels and longer computation times. Model-based OPC is an iterative solution involving repeated simulation and modification of geometry to incrementally improve the image. Managing the large geometry database is CPU intensive, and the simulations involved in model-based OPC are even more CPU intensive since there is no closed form solution for the optimal layout.

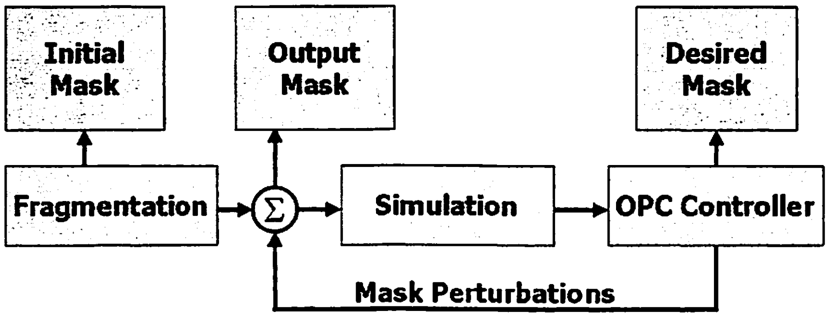


Figure 4: Model-based OPC algorithm flowchart (diagram derived from [10])

Pattern matching is similar to OPC since it involves predicting the effects of the lithographic process on the printed image. The algorithms are in fact related as well, as both

OPC and pattern matching can look up polygon features in a table to determine the contribution of each feature at some point in the layout. The rectangle and triangle pattern matching algorithms explained in Section 6.4 and Section 6.5 are similar to the edge table lookup used in the OPC algorithm developed by Nick Cobb [10][2].

2.3 Lens Aberrations

Ideal lithographic lenses act as low-pass filters, resulting in corner rounding due to the loss of high spatial frequencies. Furthermore, lenses may also have aberrations, or deviations between the real and ideal wavefronts exiting the lens. Aberrations distort light rays so that the reduced image in the wafer is not an exact demagnification of the object. Aberrations correspond to optical path differences due to inhomogeneity in the lens material, imperfect lens shape, or incorrect alignment of optical elements in the tool, which produce phase differences in the resulting plane waves. The Strehl ratio [11] is a measure of the quality of a lens and can be as high as 0.986 in modern lithographic lenses, corresponding to at most 0.014 waves of RMS lens aberration. The human eye typically contains much larger amounts of lens aberration, which results in reduced clarity of vision.

An inherent effect of lens aberrations is that they take light from openings on the mask and redirect it towards incorrect locations on the wafer, decreasing pattern fidelity and reducing the manufacturability of the design. Even the small amounts of aberrations present in modern, high-quality optical lenses can lead to Across-Chip Linewidth Variation (ACLV)

2. http://www-video.eecs.berkeley.edu/papers/ncobb/cobb_phd_thesis.pdf

and reliability problems [12]. Common aberrations include coma, astigmatism, spherical, and trefoil. Additional background material on aberrations, their effects on imaging, and their measurement techniques can be found in [13]. Other methods for measuring aberrations include phase-edge focus monitors [14], the Litel mask[3], Dirksen's phase dots [15, 16, 17], the measurement of side-lobe artifacts on halftone masks by Hayano, Fukuda and Imai [18, 24], and Kirk's double exposure method [13].

Traditional design flows include an OPC step that modifies the polygon edges to take the optical proximity effect due to surrounding shapes into account and correct for low-pass filtering. However, aberrations are additional phase errors in the Fourier spectrum [19] and may be different for each projection printing machine. A significant amount of work is in progress to characterize the aberrations in printing equipment for the purpose of "aberration proximity correction". There is no generalized solution to reducing the effects of all lens aberrations on a layout, and it is often not feasible to run separate corrective runs on each design for every aberration of every stepper or scanner. There is a need for a system that can quickly process the entire layout and locate areas where a known set of lens aberrations will have a significant effect on the printed image. Thus, the pattern matcher was initially created as a solution to this problem.

This dissertation suggests using a multi-prong approach to aberration proximity correction. The first step is to reduce the aberrations in lithographic equipment to the extent possible, which is the responsibility of the equipment manufacturer. The next step is to actually measure the dominant residual aberrations that could not be removed from the tools.

3. http://www.litel.net

The final step is to make the layouts less sensitive to these dominant residuals through a pattern matching approach. This is clearly a complex problem involving tool manufacturers, process engineers, and circuit designers.

2.4 Image Processing & Shape Recognition

Image processing and shape recognition generally fall under computer science, computer learning, or computer vision. In many cases, the images used are photographs or video camera feeds, and the objectives of computer vision and shape recognition are to locate a certain shape or sub-image in a larger image. In some cases the objective is to determine if two images match.

If the integrated circuit layout is represented as a large image and the pattern is also represented as an image (for purely real-valued patterns), then any generalized image-processing algorithm can be used. Therefore, pattern matching is a form of image processing and image processing theory can be applied. However, most image processing algorithms are inefficient for images as large as an integrated circuit bitmap representation. The reason pattern matching is still a tractable problem is that the layout "images" have certain properties that can be utilized to make pattern matching easier. The pattern matching algorithms developed under this research are compared with standard image processing and shape recognition algorithms in Section 6.6 on the basis of performance, features, and accuracy.

3 Pattern Matching Theory

3.1 Aberration Effects

The optical lenses of a non-ideal lithographic system may contain aberrations, which correspond to optical path differences in the lens material that produce phase differences in the resulting plane waves. The pattern matching approach involves determining the geometries which are the most sensitive to a residual processing effect, which is best done through a theoretical analysis of the effect. The patterns used to locate areas sensitive to lens aberrations can be derived from the optical system's parameters and aberration fingerprints of the lenses, which provide a clear representation of the effects of lens aberrations on geometries to be printed. A perturbational analysis [20] of lens aberrations is given below and also provided in [21], and a discussion of the modeling of PSM optics is given in [22]. Portions of the following discussion have been taken from [23].

An integral over the lens pupil is used to determine the optical path difference (OPD) due to aberrations and takes the form of e^{jOPD}. When aberrations are small, the exponential factor can be linearized [24] by a Taylor series expansion as $e^{jOPD} \approx 1 + jOPD$. The constant 1 gives the electric field for an unaberrated image, and the jOPD term represents an additional electric field component that spills outside of the image of a point source and is proportional to the size of the OPD. For a modern lens with a Strehl ratio [11] of 0.975 [25], a two-term Taylor series approximation is reasonable as the total RMS aberration is 0.025 waves, the peak OPD function values are about 0.05 RMS waves, and a third term is at most only 10% as large as the first term. These spillover electric fields from adjacent features must be added, and in forming the composite it is equivalent to simply determine the similarity of the region surrounding an observation point to the spillover function. This spillover function is the IFT of the OPD function and for the special case of a single Zernike [20] aberration it is the IFT of that Zernike term.

The goal is to determine the additive electric field of the jOPD term from a collection of mask openings in a neighborhood of a central observation point. One approach is to compute the contribution to the electric field from each of the surrounding pixels and then sum them up. A more interesting alternative is to first view the problem in the pupil of the lens and attempt to maximize the spillover from the jOPD term onto the unaberrated image term. In this view the additive field will be largest when the incident electric field is uniform in magnitude and exactly cancels the phase of the OPD. That is, the additive field in the pupil is proportional to $e^{-jOPD} \approx 1 - jOPD$. The inverse Fourier transform (IFT) of this function in the pupil can be used to determine the pattern on the mask that will create this maximized spillover onto the unaberrated image of the central pixel. The IFT of the constant term corresponds to a fixed infinitesimal pinhole at the pattern center. The effect of this pinhole is independent of the level of aberrations and so it may be disregarded in studying the additive perturbation due to aberrations.

The IFT of the second term yields the desired composite pattern centered at the observation point on the mask that will produce the greatest spillover onto the observation point for the given set of aberrations making up the OPD. This pattern is zero at the observation point itself due to the fact that the Zernike functions other than the zeroth that are included in aberration measurements individually have zero area when integrated over the pupil. The zeroth order term can be viewed as producing the unaberrated image complete with optical proximity effects. The contribution of the IFT test pattern at the wafer to an additive aberration-induced electric field E_A at the central observation point can be calibrated as follows. First compute the IFT for a given jOPD and digitize it into a pattern surrounding the central observation point. Then simulate the aerial image of this pattern in the presence of aberrations and take the square root to convert intensity to electric field. Here E_A is a complex quantity and its imaginary part comes from even aberrations (such as defocus, spherical, and astigmatism) while its real part comes from imaginary aberrations (such as coma and trefoil). Simulating the image of this pattern under the illumination conditions utilized in printing the wafer is believed to also help account for the reduction in sensitivity with partial coherence. The theory above implicitly assumes coherent illumination rather than the partial coherence used in various illumination schemes in projection printing.

The intensity increase and feature position change can be estimated from the electric field spillover by applying a second perturbational argument developed for assessing defect printability [26, 27]. At an observation point such as a line edge or line end, the unaberrated image of the feature can be taken as the dominant electric field contribution. The aberration spillover then adds a second smaller contribution Δ. Say, for example, Δ is 0.1. If the spillover is in phase, the intensity will increase as $(1+ 2\Delta + \Delta^2) \approx (1+2\Delta) = 1.2$. However, if the addition is in quadrature, the increase will be only $(1+ \Delta^2) = 1.01$. It is anticipated that the former case will occur for odd aberrations and the latter case will occur for even aberrations. The feature position change can then be evaluated by simply assuming that the entire feature intensity shifts upward locally, and then finding the new position of a fixed threshold image intensity.

A method of determining the intensity change and line edge shift from the electric field change is given in Section 3.3. This method is based on the procedure described in [21] of separating the portion of the electric field into components that are orthogonal to the feature electric field and co-linear with the feature, and by using the fact that intensity can be approximated by the square of the electric field: $I = E \cdot E^* = |E|^2$, where * refers to the complex conjugate operation.

A final theoretical note is that the partial coherence of the illumination plays an important role. In making the calculations for this research, the mutual coherence between the layout contribution point and the match point has been included in summing up the pattern match factor. The pattern is multiplied by the mutual coherence function for a conventional illumination system as a preprocessing step:

$$u_{TOP\_HAT}(r) = \frac{J_1(2\pi\sigma r)}{\pi\sigma r} .$$

This effect was included to be more realistic in our assessment as the more incoherent the illumination, the less the spillover and the lower the aberration impact. In so doing we also found a numerical advantage in that the error in truncating the radial integral to a finite size became more acceptable at smaller areas of integration. Results for several illumination functions are available [28].

Examples of finite radius approximations to these spillover functions for five Zernike terms are shown in Figure 5. These are actually Zernike monitor patterns and include a separate central phase reference probe for measuring aberration levels [29]. These targets consist of alternating regions of 0- and 180-degree phases, with either a 0-degree or 90-degree probe in the center. The colored phases and positions of the concentric rings and other shapes are computed using the sign of the IFT of the Zernike polynomial, and each region is separated by a small chrome border. The value of λ for this figure was chosen to be 0.5μm and NA was set to 0.5, so that λ/NA = 1.0μm. Each of the five dark boxes containing the targets in Figure 5 is 6.4 λ/NA by 6.4 λ/NA in size, or 6.4μm square.

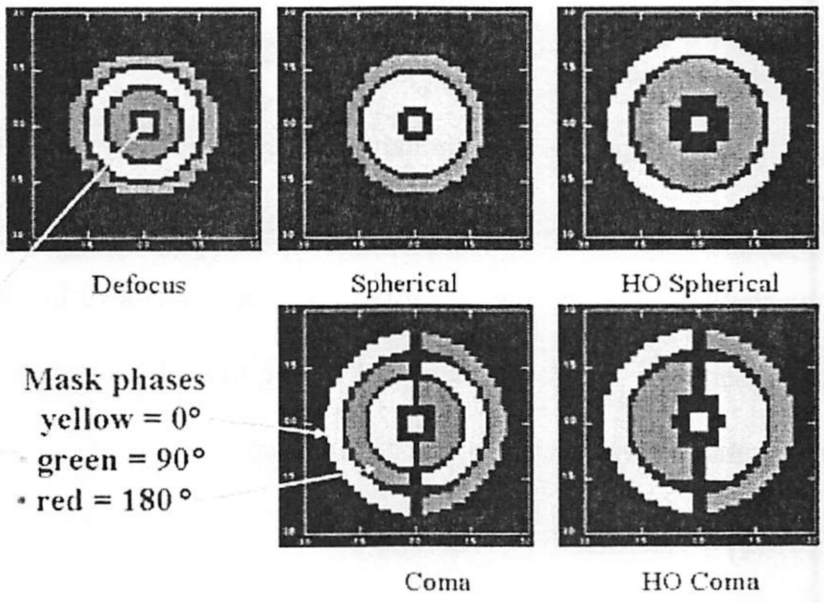


Figure 5: Zernike aberration targets (image taken from [29])

It is difficult to predict in advance whether binary or phase-shifting masks (PSMs) will be more prone to aberration effects. Since PSMs have the ability to drive both the positive and negative areas of the spillover function, it is only natural to expect phase-shifted layouts to generate match factors that are potentially twice as large as those of binary masks. On the other hand, the use of phase-shifting regions can result in steeper image slopes, especially in the case of phase-edge masks. Another important factor is the subset of feature attributes of greatest concern to the designer and technologist, which determines the range of image slopes.

The actual Zernike polynomials used to generate match patterns must be determined from the aberrations present in a specific machine or common to a family of machines, and can be measured with programmed probe-based targets [30] as well as several other methods such as that of Garza [31].

3.2 Aberration Pattern Generation

The pattern generator reads a weighted sum of Zernike polynomials as defined in Born and Wolf [20] describing the aberrations present in the optical printing machine or family of equipment of interest. Each Zernike term is in the form of a list of coefficients representing powers of rho, sin and cosine coefficients, and coefficients for phi in both the sin and cosine terms. Zernike polynomials are an orthonormal set that together specify an exact aberration fingerprint. This allows a set of aberrations present in the lens of a particular stepper or other printing system to be summed into a single pattern matrix. In fact, designers can maintain a collection of matching patterns, one for each machine, and select for the matching run the pattern corresponding to the machine(s) intended to be used in printing the design. Since flare is essentially the combination of high-order aberrations, flare components can be modeled in a similar way. The pattern generator then uses these Zernike polynomials to compute the aberrated pupil function in a circular area superimposed on a large rectangular background matrix of zeros for isolation. It then takes the inverse Fourier transform (IFT) of the sum of Zernike terms, and the center portion of the resulting matrix is written out as the pattern matrix of complex numbers. The 2D IFFT computations are implemented with a publicly available FFT package[4] and take only a few seconds per pattern.

The pattern generator reads a number of optical parameters describing the radius of the lens pupil, the size of the background matrix, and the size of the output pattern matrix. The effect of the illumination source can also be included by initializing the matrix to the shape of the pupil before it is multiplied by the Zernike terms. Partial coherence is dealt with inside the pattern matcher itself and not in the pattern generator. The pupil size is based on the wavelength of light and numerical aperture (NA) of the system. A larger background matrix provides a more accurate IFT but requires a larger IFT time and more memory. A background matrix eight times the size of the output pattern matrix is usually sufficient.

The output pattern radius is chosen so that at least 95% of the pattern weight is included in that radius, so that the pattern includes almost the entire aberration fingerprint and is a good approximation of the worst-case geometry due to that combination of lens aberrations. It is important to choose the proper radius of influence of the pattern to be large enough to capture the influence range of the aberration yet small enough to avoid excessive computation time during the matching process. A typical pattern size for λ=0.5μm, NA = 0.5, 50nm pattern matching grid is 128 by 128 pixels, which is 6.4 λ/NA on a side.

3.3 Predicting ΔE and ΔI from ΔX for Aberrations

The pattern matcher can be used to predict the change in electric field due to lens aberrations with high accuracy, as discussed in Section 8.5. The actual change in electric

4. http://momonga.t.u-tokyo.ac.jp/~ooura/fft.html

field at a feature edge is dependent on the electric fields due to the unaberrated feature, the unaberrated optical proximity effect due to surrounding features, the aberrated feature, and the aberrated optical proximity effect. These individual components cannot easily be separated, but some approximations can be made, and simplifying assumptions of mask phase and aberration type will aid in the derivation of the intensity change. The following derivation of the intensity change and line edge shift is an extension to the theory presented in Section 3.1.

After the electric field change is computed, the intensity change must be computed. There are three electric field vectors: Ea, Ep, and R. Ea is the electric field due to aberrations (both from the feature itself and from other adjacent features), which can be determined from the match factor. Ep is the electric field due to the shape that the pattern match lies on, which is in phase with Ea for a simple 0/180 degree phase match with an odd aberration. R is the electric field proximity effect of other shapes, which can be approximated from the match factor of the unaberrated pattern. The vector magnitude of Ep + R is equal to I, the unaberrated intensity at the match location. The vector magnitude of Ep + R + Ea is equal to the aberrated intensity Ia. The intensity change can be calculated with the following equation:

$$\Delta I = Ia - I = mag(Ep + R + Ea)^2 - mag(Ep + R)^2$$

$$= (re\{Ep\} + re\{R\} + re\{Ea\})^2 + (im\{Ep\} + im\{R\} + im\{Ea\})^2 - (re\{Ep\} + re\{R\})^2 - (im\{Ep\} + im\{R\})^2$$

If using a 0/180 degree PSM and an odd aberration such as coma, then Ep, Ea, and R are real, and this equation simplifies to:

$$\Delta I = (re\{Ep\} + re\{R\} + re\{Ea\})^2 - (re\{Ep\} + re\{R\})^2$$

$$= Ep^2 + R^2 + Ea^2 + 2 * Ep * R + 2 * Ea * R + 2 * Ep * Ea - Ep^2 - R^2 - 2 * Ep * R$$

$$= Ea^2 + 2 * Ea * R + 2 * Ep * Ea$$

$$= Ea^2 + 2 * Ea * (R + Ep)$$

Assuming a small amount of aberration, Ea is small and the Ea^2 term can be neglected:

$$\Delta I = 2 * Ea * (R + Ep) = 2 * Ea *_{Eunaberrated} = 2 * Ea * \sqrt{I_{unaberrated}}$$

Finally, the line-end or line-edge shift is computed from the intensity change, and that is used to calculate the critical dimension (CD) change. The line edge shift can be calculated with the following equation:

$$\Delta X = \Delta I / image\_slope = 2 * Ea * \sqrt{I_{unaberrated}} / image\_slope$$

The CD change of a vertical line can be calculated as:

$$\Delta CD = \Delta X_{left\_edge} - \Delta X_{right\_edge}$$

$$= 2 * (Ea_{left} * \sqrt{I_{unaberrated\_left}} + Ea_{right} * \sqrt{I_{unaberrated\_right}}) / image\_slope$$

Ea can be directly calculated from the match factor. $I_{unaberrated}$ can be computed through an aerial image simulation in SPLAT, or can be approximated by computing the match factor of the unaberrated pattern at that location with the pattern matcher or through fast image simulation as is used in model-based OPC. The image slope can be simulated with SPLAT or approximated based on the type of geometry (edge, line end, or corner), mask phases, and optical parameters. Note, however, that the derivation of these equations assumes a purely real phased mask and odd aberrations. The equation for imaginary phased masks and even aberrations can be calculated in a similar manner, but the equation becomes much more complex with combined real/imaginary phased masks or combined even and odd aberrations.

3.4 Match Factor Definition

The match factor is a measure of the similarity of the layout geometry to the pattern at a particular location on the mask. The match factor lies in the range [-1.0,1.0], with 1.0 being a perfect positive match, -1.0 being a perfect negative match, and 0.0 having no resemblance to the pattern. The match factor is calculated through a normalized 2D discrete correlation computation, which is similar to an image convolution. The match factor (MF) at layout position (i, j) for an X by Y pattern is calculated as:

$$MF(i+\frac{X}{2}, j+\frac{Y}{2}) = \sum_{Y}\sum_{X} Layout(x+i, y+j) * Pat(x,y)$$

In the general case both the pattern and layout values are complex numbers, so the actual result of the raw correlation computation is also a complex number. Normalization consists of dividing the raw correlation number by the best possible match factor given the pattern and set of available layout layers, and then choosing the real value, imaginary value, or magnitude of the resulting complex number, depending on the application.

The match factor is defined so that superposition can be used to independently add linear contributions from each shape overlapping the pattern. Since the correlation computation is performed on a discrete grid, diagonal and curved edges must be approximated on the pixel grid. Since each shape that overlaps the pattern may contribute some value to the final match factor, the time taken to compute the match factor is dependent on the number of overlapping shapes. Thus, the match factor is only efficiently computed when the pattern is much smaller than the overall set of polygons that form the entire layout on which the pattern match is run. This limits the applications of pattern matching where this match factor definition can be used, but fortunately there are ways to speed up the match for large patterns by reducing the resolution. The match factor at any location is independent of previously computed match factors, so only a subset of the total number of layout locations can be processed without loss of accuracy. Finally, with a typical grid size equal to a fraction of a minimum feature size and a small number of layout layers, the layout "image" consists of large groups of the same set of only a few unique pixel values. This observation can be used to speed up the naive correlation computation, as explained in Chapter 6.

3.5 MLTP Definition

A Maximal Lateral Test Pattern (MLTP) is based on an inverse function representing the worst-case geometry with respect to the processing non-ideality of interest. Multiple MLTPs can also be combined together as kernels in a larger pattern. MLTPs are generally determined theoretically through perturbational analysis. For optical effects, the MLTP is the geometry that results in the greatest spillover of light from surrounding shapes to a central point. If the processing effect of interest is lens aberrations, then the MLTP is the inverse Fourier transform (IFT) of the aberrated pupil function as described by Zernike polynomials. The MLTP is defined over a radius of influence, which determines the number of shapes that affect the results of the match with that pattern. Pattern matching can be applied to any MLTP where the sensitivity of the layout to the processing effect is proportional to the similarity of the actual mask geometry to the worst-case geometry. In other words, the higher the match factor, or correlation between the pattern and the layout, the more severe the impact of the processing effect on the printed layout at that point. This definition of match factor allows superposition to be used to sum up the contributions of each layout geometry component to the match.

The match factors are rank ordered so that the user is presented with the best N matches, which represent the areas of the layout that must be examined for susceptibility to the processing effect. These areas can then be simulated more rigorously with a process or lithographic simulator such as SPLAT [32]. If necessary, the reported geometry may be modified to reduce its sensitivity to the process effect. This process can be repeated until all of the problematic geometry is modified so that no geometry resembles the pattern. The final design will thus be robust to that particular processing effect as modeled by the MLTP. If the geometry cannot easily be modified, then these areas can be inspected more thoroughly at inspection time to determine if the process effect did cause a problem at the match locations.

4 Software System

A block diagram of the pattern matching software system is shown in Figure 6 and is explained below. The complete software system consists of about 60,000 lines of C and C++ code divided into a number of modules. The pattern generator reads a set of Zernike polynomials representing an aberrated pupil function for the optical system of interest, or any other user-defined equation, in order to generate the pattern bitmap of complex numbers. The main pattern matcher executable then reads the pattern, a user input parameter file, and a possibly multilayer mask layout in CIF, GDSII, or custom pattern matcher format. The pattern matcher is capable of reading GDSII and CIF layouts directly, storing them in compact hierarchical form for efficient access to geometry. The pattern matcher is run, and a resulting sorted table of the highest match factors and their locations for each pattern is output as well as JPEG images of the patterns over the match locations. An arbitrary number of patterns can be used in one matching run and several layouts can be combined and run together.

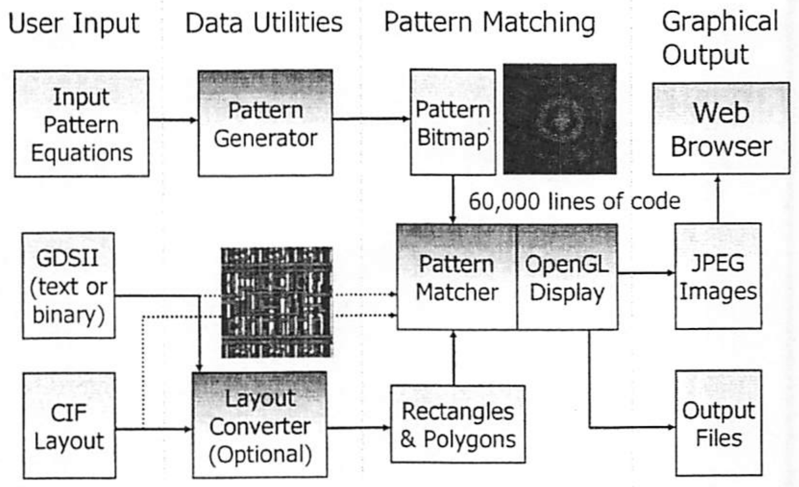


Figure 6: Pattern Matcher Block Diagram

In addition to the main matching programs, the system includes a tool for flattening and converting CIF and GDSII layouts into the internal layout format of the pattern matcher. The pattern matching software also provides a graphical interface for the user to view the layout with the highest scoring match locations highlighted. This software has been developed in several forms as discussed in Chapter 5: a UNIX version integrated with the Cadence DF II CAD tool, a standalone Microsoft Windows PC version with OpenGL GUI support, and several versions of a public web interface to the system. This web version allows remote users to input their layouts and aberrations into the system to take advantage of the full power of the tool.

4.1 Input and Output Files

The pattern matcher system first reads a set of patterns, a mask layout, and a configuration file written by the designer. The pattern is scanned over the layout in order to compute the match factors at locations of interest along edges, line ends, and/or corners, and the match factor results are sorted and written to an output file. The system generates SPLAT files of the geometry overlapping the pattern at the highest scoring locations, and JPEG, RAW, or BMP output images of the match results. The software also includes pattern generation and layout conversion utilities, and an interactive graphical user interface for viewing the match results. The various input and output file formats are described in the following subsections.

4.1.1 Patterns

The pattern matcher reads a set of patterns corresponding to a representation of the geometry on which to match. The closer the layout at a location matches the pattern, the higher the match factor. A pattern file is a binary or ASCII text 2D matrix of complex numbers stored in rectangular form. The magnitude of the complex number encodes the weight of the pixel such that larger magnitude numbers have a greater effect on the match factor. The phase of the complex numbers, which is not used in all pattern-matching applications, relates to which layout layer matches at that location.

In the case of lens aberrations, these pattern matrix values are the direct output of the IFT of the Zernike polynomials describing the aberration(s) of interest. The pattern matcher provides the option to trim the pattern values to a circle for more pleasing visual results and to remove the orientation bias. In addition, an aberration pattern can be multiplied by the partial coherence function in order to model a partially coherent system. The pattern matcher is also capable of transforming the patterns through rotations and mirroring in order to produce orientation-independent match results. Typical pattern size is 128 by 128 pixels, which corresponds to a few microns or several minimum feature sizes. Patterns are usually square and a power of two in size, but are actually only required to be at least four by four pixels and even in size.

4.1.2 Layout

The software system reads an integrated circuit mask layout in one of several geometry formats. This layout defines the geometry that the pattern will be matched to. The system includes a tool that converts hierarchical GDSII and CIF layouts into flat pattern matcher binary layouts.

4.1.2.1 Pattern Matcher Format

Pattern matcher geometry files can be stored in either binary or ASCII text formats, both of which support polygon, rectangle, and circle geometric primitives. The text geometry files support a weak hierarchy that includes scaling and translation of geometry, which is implemented through includes of either binary or text files. Geometry is specified in unsigned integer pattern matcher grid units and is referenced from the lower left corner of the layout's bounding box. Integers are used so that transformations can be performed with exact math and so as to reduce the size of the vertex data compared to double-precision floating-point numbers. Each set of geometric primitives is specified by a set of vertices and other measures describing the primitive, as well as an integer representing the layer index of the shape.

4.1.2.2 GDSII

The Graphical Data Stream II (GDSII) format [33, 34] consists of a collection of records describing the hierarchical geometry of an integrated circuit layout, including cells, geometric primitives, text, layers, and other commands. Geometric primitives include polygons, paths, and boxes represented with integer coordinates. GDSII layouts can be stored in both binary and ASCII text forms, with binary format being more compact on disk and text format being human readable. In addition, the Key is a partially supported text layout format similar to GDSII. Binary GDSII is the industry standard layout format for large designs due to its streaming nature, efficient data storage, and machine-independent format. The pattern matcher executable can read zipped GDSII binary files as well as unzipped files since the zipped data is usually about 5.5 times smaller on disk. It is also possible to have the pattern matcher read the newer OASIS layout format standardized by SEMI[5] which claims to have a 5-10X smaller data representation than GDSII, but this task is left for future work.

4.1.2.3 CIF

The Cal-Tech Intermediate Format (CIF) layout format [35] is common among academic institutions due to its human readability. CIF is an ASCII text hierarchical description of integrated circuit geometry that includes primitives similar to those in the text version of GDSII. CIF uses an integer-based coordinate system and scaling by fractional ratios for exact coordinate generation. Though CIF is not widely used for large industrial designs, many university lithography simulators read and write this layout format.

4.1.3 Layers

Each pattern matcher layer has a complex-valued weight and additional information such as color for display purposes. The layer parameters are read from the layer section of the main input file, and the complex numbers are specified in rectangular form. In the case of lens aberrations and phase-shifting masks, the layer's weight is usually 1.0 and the layer's phase is the same as the mask phase. The purpose for the layer magnitude is to assign different weights to each layer so that the importance of each layer can be accounted for. The pattern matcher is optimized for processing a single layer but can handle an arbitrary number of mask layers. The layers do not need to be for the same mask; polysilicon and several metal layers can all be matched on in a single run. In addition to the standard input layers, the pattern matcher allows an arbitrary number of Boolean and dependent layers that are

5. http://wps2a.semi.org/wps/portal

computed from the set of input geometry layers. For example, the user can define a new "transistor" layer that is equal to the areas where polysilicon overlaps active.

4.1.4 Match Requirements

Match requirements are included as sections of the main pattern matcher input file and are generally specified per pattern. The first type of match requirement is geometry-based. The pattern matcher can be constrained to only match on locations at inside corners, outside corners, edges, and line ends (edges less than a user-defined length). Edges and line ends can be further refined by requiring, for example, that only edges between two inside corners be matched on. The match locations can also be constrained by match region bounding boxes and coordinate lists.

Matches can be constrained to lie only on certain layers. Required match layers can be used to specify matches that only lie within certain regions defined by a boundary layer. Layer requirements can also be used in conjunction with Boolean layer operations to specify that all matches occur at locations where polysilicon overlaps active.

Finally, the pattern matcher supports per-pixel pattern requirements such as required layers, required corners, required edges, and required inequalities involving layer weights under specific pattern pixels. This allows the user to specify, for example, that all matches of pattern P have layer 0 at the left corner of the pattern, a left polygon edge at the left side of the pattern, and a total underlying layer weight greater than 2.0 under pixel (23, 35). In summary, match requirements allow an almost unlimited possibility of constraints to be placed on the pattern match locations.

4.1.5 Output File Matches

Pattern matching results include both a text output listing the match locations and optional images of the match location geometry. The text results file contains a sorted list of the top match locations found for each pattern. The number of match locations reported is limited by a user-defined maximum and/or a user-defined lower threshold cutoff based on a fraction of the best match found, whichever limit is reached first. Multiple orientations of the same pattern may be either listed separately or grouped together in a single sorted list, depending on which the user prefers. Each line of the results file lists the X,Y location of the match, the match factor normalized to [-1.0, 1.0], the type of match (edge, line end, inside corner, outside corner, or other), and the sum of the complex-valued weights of the layers under the match location.

4.1.6 Graphical Images

The graphical interface and resulting images allow the user to browse through the match locations and observe the geometries of interest. If the pattern matcher is run in command line mode with graphics enabled, then a number of user-selectable JPEG, RAW, or Windows bitmap (BMP) images are generated. One image shows the entire layout with all match locations highlighted and additional zoomed-in images display a user-defined number of the top match locations. These images are suitable for inclusion in web pages, papers, and presentations. An additional option to feed ASCII image capture commands to the pattern matcher through the command line allows for a pseudo-interactive interface for use in web applets. Alternatively, the pattern matcher can be run in true interactive graphical mode as described in Section 4.5.

4.2 Core Pattern Matcher

At the heart of the system is the core pattern matcher executable, the component requiring the most research and programming effort to complete. This module links the system together by reading the layout, the patterns, the layer file, and the user configuration/match requirements file. It outputs all of the match results, extracted geometry, and images. The core pattern matcher has evolved from a collection of Cadence Design Framework II SKILL procedures to a standalone executable capable of reading industry standard layouts and providing a fully functional graphical user interface. The main pattern matching algorithms used in this component of the system are discussed in Chapter 6, and details on the supporting algorithms are discussed in Chapter 7.

4.3 Pattern Generator

The pattern generator is a separate executable that can be used to generate both real-valued and complex-valued patterns for various applications. These patterns are generated once from optical parameters such as illumination source, lambda, and NA and stored for repeated use in the pattern matcher. The pattern generator has the capability of generating aberration patterns, flare patterns, and patterns based on arbitrary 2D equations. Aberration pattern generation was described in Section 3.2. Flare pattern generation is algorithmically similar to aberration pattern generation and is discussed in Section 9.1.2.

The main pattern generator program was only designed to generate lens aberration and flare patterns, but some other applications of pattern matching require patterns of a different type. An additional utility, fg_equation, complements the pattern generator's capabilities by allowing the user to input a custom pattern equation that is later converted to a pattern matrix for use in the pattern matcher. Fg_equation reads an equation file much like a set of MATLAB equations, and supports a large variety of math operations, complex numbers, discontinuous equations, and 1-, 2-, and 3-D equations in either a Cartesian or spherical coordinate system. It can be used to generate patterns for just about any closed-form equation.

A Cadence SKILL script can be invoked to perform a conversion from Cadence layout to a pattern file. This allows the user to design a custom pattern that the pattern generator is incapable of producing from input equations. For example, the user can create a pattern layout that meets design rules for a particular technology and then print that pattern on a wafer for testing purposes. The layout to pattern converter was created to allow the designer of such a custom target to run the pattern matching software so that the target pattern matches the hand crafted layout exactly. This converter can also be used to transform an arbitrary section of mask geometry into a pattern so that the pattern matcher can be used to match non-aberration based patterns to layout, effectively implementing a generalized geometric search procedure.

4.4 Layout Import/Converter

Integrated circuit layouts in CIF and GDSII formats can be processed in one of two ways. The first way involves reading in the layout and converting it into a flat pattern matcher formatted layout through the cif_to_pm utility. The pattern matcher then directly reads this format and performs the pattern matching on that layout representation. Of course, the large flat layout representation might not fit into memory and thus may need to be partitioned and written to disk in geometrically sorted order. This method only works when the layout data fits within the addressable range of the file system, with a maximum size of 2GB.

An alternative and typically much better way of processing a hierarchical layout is by directly reading and storing the hierarchy, and only flattening small pieces of the layout just prior to the matching of the pattern on that piece of layout. This avoids storing any intermediate data on disk and usually reduces both the preprocessing runtime and memory required to run the pattern matcher. The layout is also compressed in memory and spatially subdivided for more efficient spatial queries. The internal hierarchical database format is the same as in simpl_display, which is explained in Section 4.5.2.

4.5 Interactive Display Tools

Several OpenGL-based display tools are included in the software system for interactive viewing of the input files, pattern matcher results, and the outputs of various simulators developed by the UC Berkeley EECS Department. The tools are for the most part cross-platform and have been used in both Windows and UNIX. These viewing tools are discussed below.

4.5.1 Pattern Matcher Graphical Output

If run in interactive mode with graphics enabled, the user is presented with a display window containing the layout with patterns and text overlaid to show the match results. The user can then use pan and zoom commands, make measurements, take screenshots, and query objects in the layout. The supported features include those in simpl_display, which will be discussed in the next section. In addition, the interactive pattern matcher allows the user to automatically zoom to match locations and to view the pattern images under various graphical settings.

4.5.2 Simpl_display

Simpl_display is an industrial strength CIF and GDSII layout viewer for large integrated circuit layouts. It utilizes OpenGL for efficient, hardware-accelerated display of large polygon datasets, including measurement, object query, image capture, and GDSII write functionality. Simpl_display has been designed with import, query, and display performance that is competitive with commercial EDA tools. Simpl_display and the graphical pattern matcher share much of the same code base and therefore have nearly identical display and query functionality.

4.5.2.1 Cell Data Structures

Hierarchical layouts are stored as an array of cells, where each cell contains a collection of rectangles, polygons, paths, circles, text, and references to other cells. The use of hierarchy allows a design such as a large memory chip to be represented very compactly by storing a single copy of a memory cell and an array of references to that cell, which together make the entire matrix of memory cells. The geometry inside of the cells includes vertex arrays, bounding boxes, layer identifiers, and other information needed to iterate over and perform efficient queries on the data.

4.5.2.2 Shape Compression

Full chip GDSII layouts can contain more than ten gigabytes of data, even with substantial hierarchy. These layouts are difficult to fit into memory without data compression, and this is an even greater problem when the layout data exceeds the address space of 32-bit machines. Shape data compression is needed to reduce the size of the database in memory (or on disk if the database is stored in virtual memory or as a file). There are a number of different ways to store cell geometry and instance information in a lossless, compressed form. One method involves low-level compression of the layout raster image [36], but compression at the cell and polygon levels seems more appropriate when using the latest rectangle and triangle algorithms.

The GDSII format itself has a number of storage inefficiencies [36, 37]. For instance, the first point of a polygon is repeated at the end of the polygon. Boxes, or rectangles, are described with four vertices when only two are necessary. Instances reference a cell by the cell's name and not by the cell's ID. Some record types contain extra data and unused bits. All of these simple storage problems are remedied in the pattern matcher cell database.

Many layouts contain duplicate polygons where the only difference between one polygon and the next is a translation. This is especially true for repeated geometry such as memory cells in flat layouts. Each time a polygon is read, the GDSII parser computes a hash value of the polygon edges and checks for the existence of a polygon with the same shape. Each set of compatible polygons is stored as an array of polygon translations and a single vertex array. This polygon data compression idea can be extended to rectangle compression and to include polygon rotations and mirroring, as described in [38]. Polygons from other sub-cells of a spatially subdivided cell can also point to common vertex sequences. Cell names and text strings are hashed in a similar manner so that text data can be reused.

Simpl_display and the pattern matcher maintain tables of common cell transforms so that most of the instance transformations can be replaced with a single integer corresponding to an index into the transformation lookup table. The default table contains all combinations of rotations by 90-degree increments and mirroring transforms, for eight basic transformation sequences. Translations are rarely repeated a large number of times, so they are not included in the lookup tables. Using this system, most instances can be compressed down to a cell ID, X and Y integer translations, and a transformation index or pointer, totaling only 16 bytes.

Many layouts contain mostly rectangles, and thus the polygon compression tricks cannot be used as effectively. Rectangles are represented by 32-bit integers in the input GDSII, but the cell database separates these rectangles into large (32-bit) rectangles of 20 bytes, small (16-bit length and width) rectangles of 16 bytes, and small (16-bit width) squares of 12 bytes. Since most of the rectangles in a layout are small and most of the contacts and vias are squares, this optimization achieves an additional compression ratio of about 1.5.

GDSII supports arrays of cells, but not all tools take advantage of arrays when generating a hierarchical GDSII file. The cell database described here automatically combines instances of the same cells into arrays and arrays into two-dimensional matrices, which is extremely effective in reducing the database size for large memory and cache blocks.

Finally, some cells may be empty or contain zero width shapes or other non-displayable geometry. In addition, when running the pattern matcher on a layout, the user often does not care about all cells and all layers. In these cases, the unused data is removed from the cells, and any empty cells are removed from the hierarchy in a bottom-up manner so that an entire tree of unnecessary cells is removed and the memory is freed. This can yield significant database size reduction.

In one example, a 234MB GDSII file was read and optimized for minimum database size. The first step of combining instances into arrays reduced the GDSII file size to 68MB. The next step of removing unused cells further reduced the size of the layout to 17MB. The other compression methods combined reduced the database to a final size of 6.5MB in memory, an overall compression ratio of 36. Another layout was reduced from a 462MB file to 80MB of memory, mostly due to the grouping of millions of polygon fill shapes into a small number of polygon arrays.

4.5.2.3 Spatial Subdivision

In order to support efficient geometric query and display of polygons, cells should contain between 100 and 10000 objects. If there are too many objects in a cell, then the iteration time required to find all objects satisfying a particular requirement may be too high. On the other hand, cells with only a few objects incur a large overhead each time the cell structure is loaded into cache and each time a cell operation is initiated. Some flat layouts contain millions of objects in a single cell, and every object must be touched to find the object at a certain location within the cell.

Simpl_display reduces cell processing times by adaptively dividing large cells into a number of smaller cells. The smaller cells are further subdivided until the number of shapes in each cell is below a threshold. With these smaller sub-cells in place, an entire sub-cell can be skipped if the query point lies outside of its bounding box. Coupled with shape recognition and polygon hashing, duplicate sub-cells can be found and hierarchy can even be introduced into a flat layout. With the proper spatial subdivision, local query time has almost no dependence on the number of objects in a cell for medium to large cells.

For example, consider performing a single point query within a uniformly filled, flat layout containing one million non-overlapping shapes. If the layout is subdivided by a factor of one hundred each time a cell contains more than one thousand shapes, then the first subdivision will yield one hundred cells, each with ten thousand shapes. The next subdivision will divide each of the hundred cells into one hundred new cells, each containing one hundred shapes. The point query will first iterate over the initial hundred cells to find the one that contains the point, then iterate over the hundred cells within that cell, and finally iterate over the hundred shapes in that sub-cell. Using two levels of spatial subdivision, this query takes only 300 operations instead of one million.

4.5.2.4 Geometry Query

It is important to have fast query capabilities when working with full chip layouts. In many cases, the runtime of an operation is entirely determined by the total time taken by a large number of small queries. Typical geometric queries include request for all shapes on a particular layer, all shapes within a bounding box, all shapes of a certain size, and all shapes of a certain classification. In general, there is no optimal way to support all of these queries with a single database. However, compromises can be made that lead to fairly efficient times for all types of queries. The two major requirements for an efficient query are reducing memory bandwidth through shape compression and reducing search time through spatial subdivision, as discussed in the previous two sections.

The cells themselves are spatially subdivided into smaller cells, thus permitting fast spatial data queries. Polygons, paths, circles, large rectangles, small rectangles, and squares are stored separately. Shapes within a cell are first sorted and grouped by layer, allowing all cells on a given layer to be accessed efficiently. Within a layer bin, the shapes are sorted by size so that all shapes above or below a certain size threshold can be returned without iterating over the shapes that fail the size requirements.

4.5.2.5 Graphical Display

Large layouts can contain hundreds of millions of shapes on dozens of layers, and thus care must be taken to efficiently display datasets of that magnitude. Shapes that are out of the viewing region or are too small to see can be skipped over. When the layout is fully zoomed out, most of the shapes are smaller than a pixel and can be clipped; when fully zoomed in, most of the shapes are off the screen and can be clipped as well. Additionally, complex shapes can be approximated by simpler ones when small in size with little or no reduction in image quality. Each of these clipping tests requires efficient geometric query operations, which in turn require a good spatial subdivision of the layout.

Displayed primitives include rectangles, polygons, paths, circles, text labels, and cells. OpenGL allows primitives such as rectangles, convex polygons, circles, and text to be drawn on the screen. Concave polygons must first be divided into a series of triangles, triangle strips, and triangle fans. Triangulation is performed by the GLUT tessellation engine when the polygon first comes into view, and the triangle primitives are collected into display lists and drawn on the screen. The display engine caches the display lists so that they can be used for similar polygons, thus dramatically decreasing the amount of memory required for the display list triangles. Manhattan paths are drawn as a series of rectangles, and circles are used to represent the corners of rounded paths. Non-Manhattan paths are first converted into polygons and then drawn. Self-intersecting paths and polygons are supported since extra vertices can be added to break them apart into simpler polygons. The final list of shapes is sorted by layer and by size so as to minimize query time.

In order to display hierarchy, the display engine traverses the cells recursively in depth-first order, clipping entire cells if they are off the screen or their contents are too small to be seen. OpenGL transformations are used to apply translations, rotations, scaling, and mirroring to the cells, and various display information is cached and collected into arrays to speed up the drawing process. Once a user-selected cell depth is reached, only the bounding boxes and names of the cells are displayed. Each cell contains iteration lists, layer tables, size statistics, and other data that is used to efficiently access a cell's geometry and to filter out cells that do not need to be drawn.

Polygons and other drawing primitives can be drawn with solid colors, alpha-blended for partial transparency, drawn as wire-frame, or filled with stipple patterns that are read from a pattern definition file. Small polygons are approximated with a combined static and dynamic level of detail (LOD) algorithm that displays only the larger features of the polygons while removing the smaller features. Even smaller polygons are approximated with one or more rectangles, and tiny polygons less than a pixel in size are drawn as alpha-blended, partially transparent pixels. Paths less than a pixel in width are drawn as alpha-blended lines.

Pan and zoom are supported by transforming the entire layout view in OpenGL. The user can select a zoom area by drawing a box on the screen with the mouse to get a closer view of the layout geometry. The pan operation, invoked with the arrow keys, supports a double-buffered bitmap copy to avoid redrawing parts of the screen that were visible before the pan, thus significantly speeding up scrolling of the layout. A small thumbnail image near the lower left corner of the screen supports a view of the full layout and an indicator of the current window position. The user can click inside the thumbnail image in order to navigate around the layout.

The graphical display includes several measurement and navigation tools. Clicking on polygons and cells will show the user numerical information about what is selected, and layers can be shown and hidden by clicking on the layers. Clicking on cells in simpl_display's open mode opens the target cell in the viewer. A scale bar is available in the bottom left of the screen, and a ruler command allows the user to measure the X and Y dimensions of polygons and cells in the layout. The user can take screenshots in a variety of color modes and image formats, including bitmap images of arbitrarily high resolution. In addition, simpl_display has the capability of writing out GDSII files of selected parts of the layout. The layout viewer simpl_display and the integrated view for the pattern matcher contain many additional features that are typically found in other viewers and CAD tools.

4.5.3 Display3D

Display3D is an OpenGL visualization utility for lithography simulator input and output files. It can read and display a number of 2D and 3D formats including SPLAT mask layouts, SPLAT contour plots, SPLAT pupil maps, STORM 3D plots, TEMPEST 3D plots, and NEtch 3D meshes. Display3D supports drag-and-drop of simulation data files and interactive viewing of solid and wire-frame models that can be panned and rotated on the screen. It was intended as an easy-to-use, hardware accelerated alternative to existing JAVA utilities such as drawmask and drawplot.

4.6 Interface to SPLAT Simulator

SPLAT [32] is an aerial image simulator developed at the University of California, Berkeley that produces image intensity plots along cutlines and contour plots of intensity over pattern areas from an input file representing a portion of a mask layout. The SPLAT file format consists of a header defining variables such as σ, λ, NA, and simulation area followed by a list of rectangles and finally plot commands. The pattern matcher executable automatically extracts the geometry overlapping the match locations and writes it to SPLAT files. The number of extracted SPLAT files and the area of the extracted regions are specified in the pattern matcher input file. This feature allows the matched geometry to be simulated more rigorously using SPLAT, and with a minimal amount of user effort. The SPLAT results can then be loaded into Excel, MATLAB, or a viewer such as Display3D to see the effects of aberrations or similar lithographic effects on the printed image.

A Cadence SKILL procedure was written to directly convert a Cadence mask layout to a SPLAT file as well. A second SKILL procedure converts SPLAT files back into layouts, thus closing the data conversion loop.

This SPLAT file extraction procedure was utilized in order to verify the pattern matcher results against electromagnetic simulation of lens aberrations, and to measure the actual impact of aberrations on the printed image. These tests led to bug fixes and improvements in both the pattern matcher and SPLAT itself. SPLAT does not have the ability to simulate all of the process effects for which the pattern matcher can be used, but extraction support for other simulators can easily be added to the pattern matcher.

The bitmap to rectangle extraction algorithm involves searching for the bottom left corner of a rectangle and expanding up and to the right while removing the rectangle pixel-by-pixel from the bitmap matrix. The algorithm proceeds from the lower left corner to the upper right corner of the extraction region, extracting rectangles until the region is empty. This algorithm is linear in the size of the extraction region and efficient, but the set of extracted rectangles is not always minimal. The newer rectangle and triangle algorithm that has replaced the bitmap algorithm already stores the necessary primitives, so the shapes are simply clipped to the pattern or extraction region area and written to the SPLAT file.

4.7 Usage of Results

The pattern matcher produces both textual and graphical output containing the match results. The text results can be loaded into another CAD tool for visualization purposes, or sent to other individuals in the design and manufacturing flow. The designer or process engineer can then take these match results and modify the identified "bad" geometry in the layout in order to reduce the sensitivity to the processing effect of interest. It may be possible to extend the pattern matching software so that it can automatically correct the layout to reduce the sensitivity, though that is out of the scope of this research. It might also be possible to use the pattern matcher concept to generate correct-by-construction geometry by modifying place and route tools so that they take the bad patterns into account when creating the layout. The match results can be used at later stages in the design flow as well, in order to direct inspection tools to examine areas of suspected manufacturing problems. This is potentially useful in cases where the flagged geometry cannot be modified.

5 Software Evolution

5.1 Cadence DFII/SKILL

The first experimental version of the pattern matcher was written as a set of SKILL procedures for the Cadence Design Framework II (DFII)[6]. SKILL is an interpreted programming language similar to both C and LISP that is common to all of the Cadence tools. The Cadence layout database was a good starting point for testing the pattern matcher theory on small test layouts, and the Cadence layout import of GDSII and graphical user interface (GUI) provided for a user-friendly initial development environment. The layout was streamed in from disk, stored as a Cadence cell view, or hand-drawn from generic drawing layers representing the various mask phases. The pattern was generated with an early MATLAB version of the pattern generator and was loaded into Cadence as a text file. The pattern matcher was then run as a SKILL procedure that read the input files, matched on the layout, and drew the patterns over the layout with a bitmap of colored rectangles. A collection of SKILL scripts that still remains in use provides a semi-automated process to convert between layouts, pattern matrices, and SPLAT aerial image simulation files.

Though the initial Cadence pattern matcher was simple and easy to use, it did not scale well to larger layouts. SKILL did not support the proper Boolean operations, complex number data-types, hash tables, and database spatial subdivision algorithms required for a fast and memory efficient pattern matching algorithm. A further limitation was the slow loop iteration due to interpreted code, and the difficulty of modifying and debugging compiled

SKILL code. Also, SKILL uses a garbage collection method to deal with memory allocation and freeing, while a more explicit memory management system was needed for the memory-intensive matrix-based algorithms. The next step was to move the core matching algorithms outside of SKILL.

5.2 Text-Based Executable

The most time-consuming parts of the pattern matching algorithm, including the polygon processing and inner match factor computation loop, were written in C++, compiled as a separate binary, and called from inside SKILL to do the heavy processing. The text-based executable provides basic pattern-matching capabilities similar to those available in the Cadence version, but with improved runtime and memory requirements. The standalone pattern matcher contains custom data structures and algorithms and is compiled, not interpreted. Graphical and layout processing capabilities are still achieved through integration with the Cadence development environment, where the executable communicates with a Cadence SKILL process through input and output files. In addition, Cadence is used to flatten hierarchy, perform geometric transformations, determine overlap, and merge shapes on various layers because of its efficient internal implementations of complex geometric operations. The non-graphical part of the system and some of the graphical parts are cross-platform and have been compiled under Windows, Solaris, and Linux on various 32- and 64-bit architectures.

---

6. http://www.cadence.com

Cadence is used to first flatten the layout and then merge the shapes into a minimal set of non-overlapping polygons. Cadence was originally used to split the polygons into rectangles, but that functionality was later moved into the pattern matcher core executable so as to reduce the file size. It then creates a large intermediate file consisting of rectangles, polygons, patterns, and parameters required for the matching algorithm. Cadence then executes the core pattern matcher, which reads the intermediate file, runs the matching algorithm, and produces two results files. The first results file is read by another SKILL script and the results are displayed graphically in the layout window. Each pattern is drawn over the layout at the match location as a bitmap color coded for phase along with a text string specifying match type, normalized score value, pattern ID, and underlying layer phase. The other results file contains extracted rectangles that can be converted into SPLAT file format by the final SKILL script.

The first working attempt at an external matching algorithm was slow and memory inefficient, although redesigned and significantly better than the original SKILL version. Further work on the code involving multi-level matrix compression, edge and corner data structures, partitioning, pre-filtering, and polygon operations resulted in greatly improved speed and controllable memory requirements. Eventually, new features and options were added to the core binary, making it more powerful and the original Cadence version obsolete.

5.3 OpenGL Graphical Interface

Cadence provides a convenient graphical interface for displaying the pattern match results, but Cadence was not designed for efficient display of many bitmap images.

Approximating the bitmap patterns as a large array of rectangles is not only slow, it results in a poor quality image that only supports the colors available in the standard drawing layers. Relying on Cadence for display purposes also assumes that the users have Cadence installed on their machines and sufficient resources to run large pattern matching jobs on those machines. This requirement forced the development to be done on a heavily loaded, remote system.

The next step towards an independent pattern matcher was an integrated graphical display. The initial target platform was Microsoft Windows (Win32), and OpenGL[7] was chosen as an easy to use, efficient, and well-supported graphics library. OpenGL is also cross-platform, so the display code will work on non-Windows systems such as Linux. This graphical interface eventually supported many more features than were possible with Cadence. The pattern matcher was ultimately merged with simpl_display's layout viewing engine for full graphical display of large hierarchical layouts.

5.4 Hierarchical Layout Import

The only dependency left in the pattern matcher was the input layout in a simple pattern matcher format. Cadence was still used to stream in GDSII and other layout formats, flatten the hierarchy, and convert the data to pattern matcher format. Simpl_display, the layout viewer initially developed for reading SIMPL-2 [39] cross sections and CIF files, provided much of the desired import functionality. The CIF layout import portion of

7. http://www.opengl.org

simpl_display was thus modified to create a layout conversion program, cif_to_pm, which reads CIF layouts and writes flat pattern matcher layouts. The addition of a GDSII parser to the simpl_display/cif_to_pm code allowed GDSII layouts to also be converted into pattern matcher layouts.

It was soon realized that flattening the CIF and GDSII layouts was not the answer to the layout import problem. The flat representation of a GDSII layout as small as 17MB can be as large as 1GB or more, and even larger layouts may exceed the address space of the 32-bit file system. The answer was to have the pattern matcher directly look into the hierarchical cell database of simpl_display and only flatten a single partition at a time. This operation was supported by the efficient data query operations of simpl_display. Eventually, the display code for the pattern matcher was modified to use the simpl_display graphics engine for drawing of hierarchical layouts. The features that the pattern matcher inherited from simpl_display were discussed in Section 4.5.2. The final version of the pattern matcher is a single standalone executable that reads hierarchical layouts, compresses the hierarchy, runs the pattern matcher, and graphically displays the match results over the layout geometry.

5.5 Parallel Pattern Matcher

Many of the machines used for simulation in industry contain multiple processors, and it makes sense for the pattern matcher to utilize all available processing resources to improve the matching runtime. Since the match factor can be computed at any point in the layout independent of the other match factors, pattern matching is an inherently parallel operation. The pattern matcher parallelization strategy for both Pthreads and MPI is explained in Section 7.8. The parallel pattern matcher has been successfully tested under Linux and AIX and was demonstrated to have near perfect scalability. The parallel algorithm based on Pthreads should provide parallelization on just about any shared memory multiprocessor machine running most operating systems.

5.6 LAVA Website

The UC Berkeley TCAD group's LAVA/Volcano website contains publicly accessible interfaces to various lithography simulators [40][8]. The pattern matcher has generated significant interest from industry, and a web version has been created to allow all users who have access to a web browser to try out the pattern matching system. Details of the web version of the pattern matcher can be found in [41]. The client/server model of the web version of the pattern matcher and the web interface itself were written and maintained by a number of undergraduate students in the TCAD research group. The pattern matcher is run on a server at UC Berkeley so as to not load the client machine and so that the source code and executables do not have to be distributed. CGI and Perl scripts are used to create and manage the session directory where the I/O files will be stored and to actually run the simulation executables. This web interface allows remote users to input their own layouts and aberration patterns into the system to take advantage of the full power of the tool.

The web tool was initially developed as a JAVA web applet and was later rewritten using Forms and PHP for a more intuitive user interface that loads more quickly and does not

8. http://cuervo.eecs.berkeley.edu

require installing special JAVA libraries. The two versions of the online pattern matcher are described below. Both versions return an image of the final pattern match full layout view and a zoomed in view of the best match location in the client's web browser. The images are compressed using the JPEG format for reduced network bandwidth and improved response time.

5.6.1 Java Applet

The first pattern matcher web interface uses a complex JAVA applet, shown in Figure 7[9]. The applet contains a list of layout thumbnail images at the top and a button for uploading user layouts. The available preprocessed layouts include simple test structures (lines, line-ends, T-junctions, etc.), basic 2D serpentines, and snippets from actual layouts. The middle section shows example pattern thumbnail images with a similar button for creating custom aberration patterns. The buttons under the patterns are used to set the match geometries of the selected patterns (edge, line end, inside corner, outside corner, etc.) Both the layouts and the patterns can be scrolled through, and newly added layouts and patterns are added to the list and stored in a unique session on the server. The bottom section of the applet contains text boxes and buttons for entering optical parameters and setting up the matching configuration. The applet allows the user to specify values for λ, NA, and partial coherence σ, but currently is not set up to scale the layout with the k1 factor. At the very bottom of the applet window is the button that submits the job to the server. After the pattern matching run is complete, the applet brings up a new window showing screenshots of the match results, with the patterns

9. http://cuervo.eecs.berkeley.edu/volcano/applications/pm/pm.html

drawn over the layout.

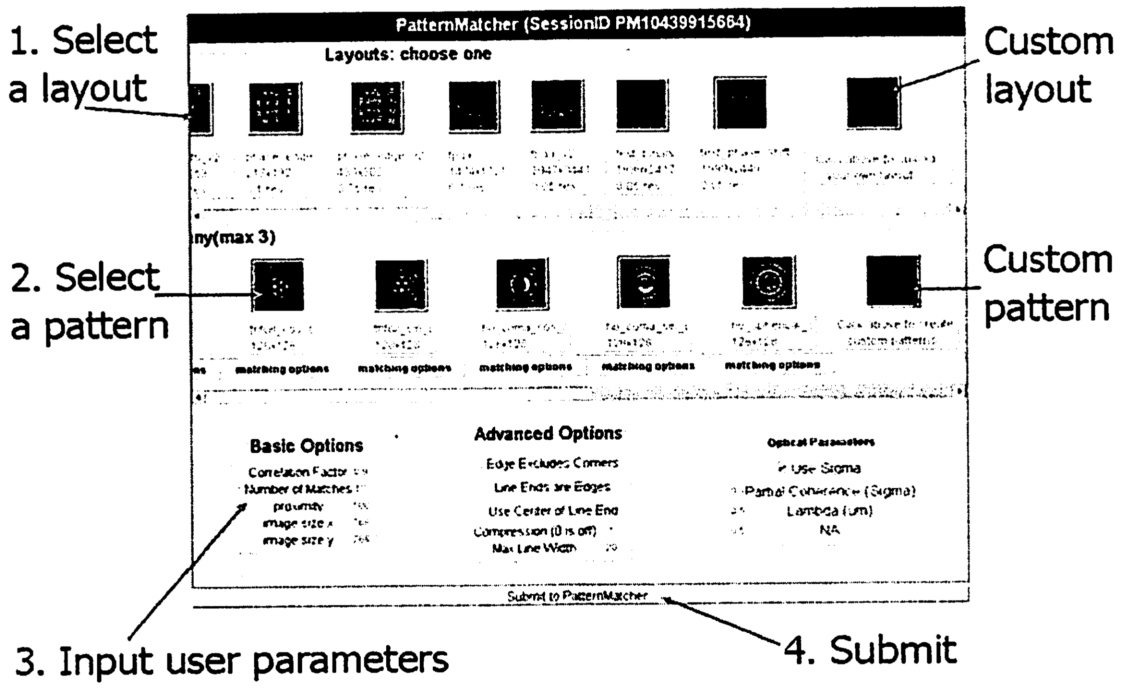
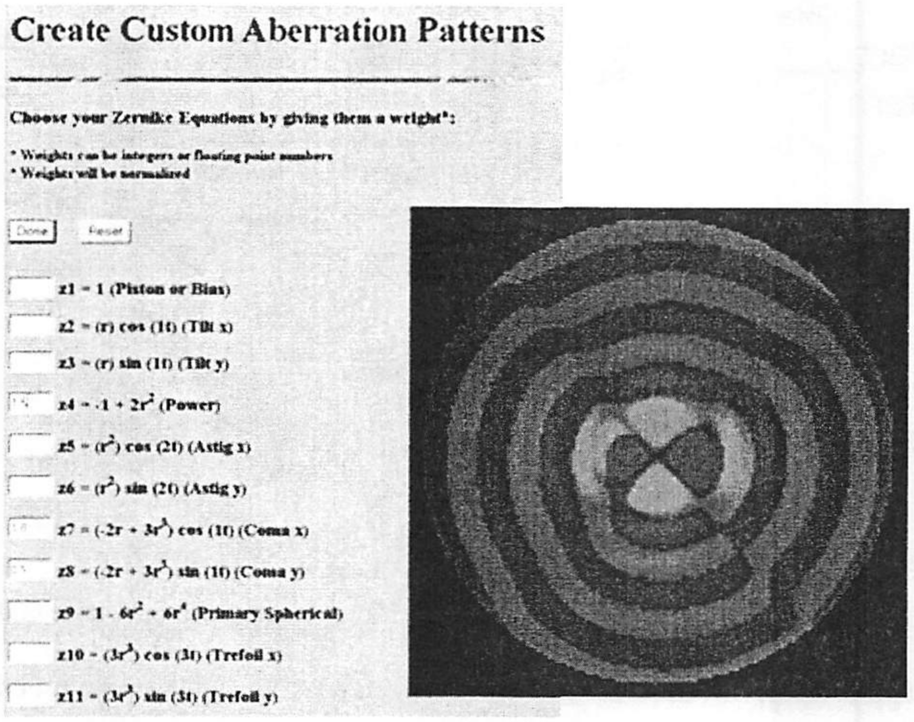


Figure 7: Pattern Matcher web applet

If the user wishes to run the pattern matcher over his or her own CIF or GDSII layout, the layout upload feature can be used. The user selects a layout and layer definition file that is stored on the client hard drive, enters a scaling parameter that defines the pattern matcher grid resolution, and clicks the "upload" button. The layout size is currently limited to 2MB in order to conserve disk space on the server, since the uploaded layouts are stored in a sessions directory for later use by the same user.

Clicking on the custom pattern button takes the user to a screen listing the first 64 Zernike aberration terms, as depicted in Figure 8. The user enters floating-point weights into the boxes of the selected Zernike terms to define the set of aberrations that make up the pattern. Clicking on "done" then runs the pattern generator on the server, which computes the IFT of the aberrated pupil function and produces a 128 by 128 pixel complex-valued pattern that can then be used in future pattern matching runs from the current session. The pattern generator also generates new thumbnail images so that the user can select the pattern by sight. Currently, patterns are limited to a fixed pixel size and fixed resolution, but the imported layout can be scaled to account for different technologies.



Create Custom Aberration Patterns

Choose your Zernike Equations by giving them a weight*:

* Weights can be integers or floating point numbers
* Weights will be normalized

z1 = 1 (Piston or Bias)

z2 = (r) cos (1t) (Tilt x)

z3 = (r) sin (1t) (Tilt y)

z4 = -1 + 2r^2 (Power)

z5 = (r^2) cos (2t) (Astig x)

z6 = (r^2) sin (2t) (Astig y)

z7 = (-2r + 3r^3) cos (1t) (Coma x)

z8 = (-2r + 3r^3) sin (1t) (Coma y)

z9 = 1 - 6r^2 + 6r^4 (Primary Spherical)

z10 = (3r^3) cos (3t) (Trefoil x)

z11 = (3r^3) sin (3t) (Trefoil y)

Figure 8: Custom pattern generation webpage and example custom pattern

5.6.2 Forms/PHP Web Application

The previously described JAVA applet has a number of limitations due to its complex and dynamic nature. The client machine must have a recent web browser and a recent version of the JAVA runtime environment installed in order to run it. The load time is long, especially over a slow network connection, and the user interface is too complex to understand without a user guide.

The newer web version of the pattern matcher shown in Figure 9 was created with

Forms and PHP for more universal web browser compatibility[10]. It loads faster and takes up less screen space. The step-by-step procedure of selecting or uploading a layout, selecting a pattern, setting up the match parameters, and executing the pattern matcher is much more intuitive then presenting the user with all of the information at once. In addition, the user can go back and change the data entered at previous steps without having to re-enter data.

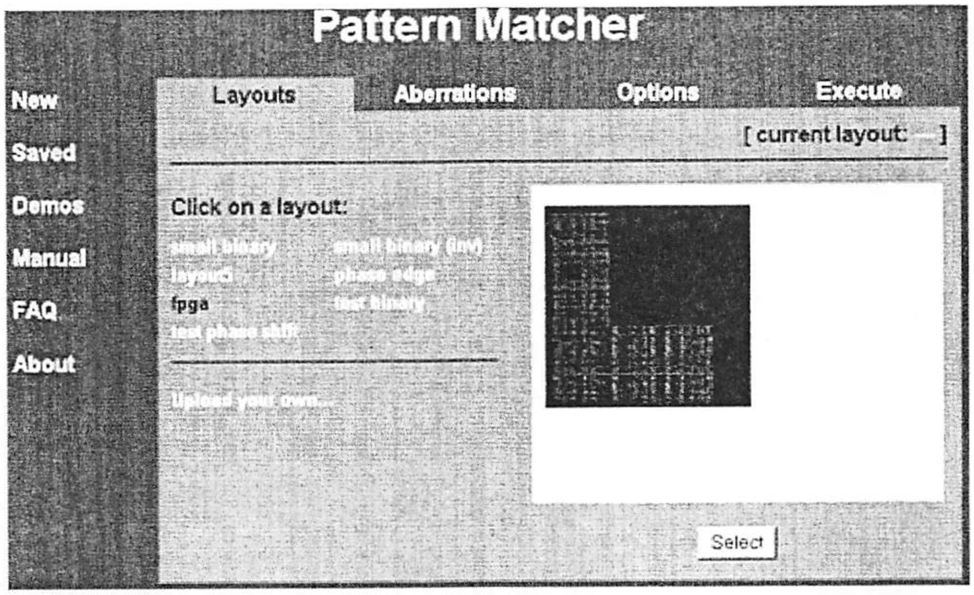


Figure 9: New pattern matcher web interface using Forms and PHP

The new web interface is incomplete, however, and has a number of limitations not present in the initial JAVA applet. For example, custom pattern generation is not yet supported, and only a single pattern can be matched per matching run. In addition, some of the links are broken. This version of the interface is currently being cleaned up and should eventually be equal in power to the JAVA applet version of the pattern matcher. It may be possible to add an interface to SPLAT so that pattern matching extracted files can be automatically directed to SPLAT for aerial image simulation. Also, if there is interest from

10. http://cuervo.eecs.berkeley.edu/pm

industrial and academic users, then a custom pattern generation system may be included so that visitors can use the pattern matcher for applications other then lens aberrations. The future web interface may even include a method for users to save their sessions and log in later to find their layouts and custom patterns saved on the server.

6 Pattern Matching Algorithms

The goal of the pattern matcher is to match a pattern to a full chip layout in under an hour on a standard desktop computer. This is a difficult challenge, considering today's layouts contain tens of millions of transistors composed of hundreds of millions of polygons and stored in hierarchical GDSII layouts of many gigabytes. The pattern matching data structures and algorithms must be able to efficiently store, access, and process the large geometric databases within a very small time per match point in order to complete the matching process in reasonable time per pattern.

This chapter describes some of the pattern matching algorithms that have been developed, including the original bitmap correlation algorithm, the improved edge-based algorithm, and the efficient rectangle and triangle algorithm. These algorithms take advantage of the unique characteristics of matching a bitmap image to a large polygon dataset: enormous search space, large groups of the same valued layout pixels, only a few unique complex-numbered pixels, opportunities for match location filtering, and low match factors. Additional information regarding the pattern matching algorithms that have been developed as part of this research project can be found in [42] and [43]. These algorithms are later compared to existing techniques in the fields of computer vision, image processing, video compression, and polygon-based matching on the basis of resource requirements and computational complexity.

The matching algorithm itself has four main steps as shown in Listing 1, and these steps are taken for each partition of the input dataset. First, the layout is read and partitioned and the pattern is read and pre-integrated. Then the input shapes are split into geometric primitives, depending on the matching algorithm. Next, the primitives are spatially subdivided in both the X and Y dimensions and sorted to permit efficient access to local areas of data. This includes partitioning the layout into smaller overlapping areas and locally flattening any hierarchy. It is possible to perform the sort and flattening operations either in memory or on disk. The primitives are sorted by Y-value and then by X-value, and may be grouped together spatially into sub-regions of a partition for fast access to local collections of shapes.

Listing 1:

```
1. Read and partition the layout, read and pre-integrate the pattern
2. Divide input shapes (polygons) into geometric primitives
3. Spatially organize primitives by x, y, etc.
4. Compute Match Factors:
      For each pattern P {
            For each X,Y match location {
                  For each geom. Primitive G overlapping P {
                        MF(X,Y) += contribution of G ∩ P(X,Y)
                  }
            }
      }
```

The fourth step is the matching loop itself, where all of the runtime is usually spent. The outer loops iterate over each pattern, orientations of each pattern, and match types (edge, line end, corner, etc.). The inner loop iterates over the match locations for each match type of each pattern orientation. The number of X,Y points tested depends on whether the user is searching all or part of the layout and if the match locations are specified to be corners, edges, line ends, or all points on a layout grid (typically 1/6 to 1/10 of a feature size). Even if the points are constrained to lie on the edges of features, the number of test locations can be in the billions for a layout of several square centimeters. Therefore, it is critical to minimize the time taken to compute the match factor at a point. The computation time per point is equal to (1) the number of geometric primitives that overlap the pattern at that point multiplied by (2) the time taken to add the contribution of a primitive to the accumulated match factor. This assumes that (1) can be determined in time linear in the number of overlapping primitives, which is guaranteed by the spatial sorting.

Four pattern-matching algorithms have been developed based on bitmaps, edges, rectangles, and triangles. Each of the pattern matching algorithms relies on the fact that the match factor is a linear sum of contributions from the weights of each of the primitives overlapping the pattern. The contribution of each primitive is independent of the others and can be added in any order. Several other classifications of algorithms can be used for pattern matching, such as Fourier Transform-based methods, but they typically do not take advantage of the large groups of similar layout pixels, small number of unique layout pixel values, and limited set of filtered query points, and are thus typically less efficient. These alternative methods are described below as well.

6.1 Primitives

When choosing primitives to use in the matching algorithm, there is a tradeoff; using higher order primitives vastly reduces (1) but also increases (2). However, the increase in (2) is more than offset by the order(s) of magnitude decrease in (1) with larger primitives. Example numbers and costs of primitives for a typical layout and 128x128 pattern are shown in Table 1. Triangle primitives have a variable computation cost since they may need to be split into as many as three separate shapes when they overlap the edge of the pattern, though the cost is usually only four operations. Edge/rectangle strip primitives are much more efficient than pixel primitives, and rectangle/triangle primitives are even better. The improved efficiency of using edges and rectangles comes at a cost of increased algorithm and code complexity and more special cases.

Primitive	**Avg. number in 128 x 128 pattern**	**Cost/ primitive**	**Total cost**
Pixel	16384	1	16384
Edge/Rect. Strip	600	2	1200
Rectangle	20	4	80
Triangle	5	4-12 (5)	20-60 (25)

Table 1: Comparison of geometric primitives used in pattern matching

6.2 Bitmap Algorithm

The bitmap algorithm is the simplest method for computing a match factor at each point, but it is also the slowest. In order to execute the inner matching loop, each input shape must be converted into a large bitmap, specifically a matrix of complex numbers representing the magnitude (transmittance) and phase of mask openings. The bitmap can be many gigapixels in size and often must be split into smaller partitions in order to fit it in a reasonable amount of memory. In addition to storing the bitmap itself, the individual polygon edges and corners are stored before the splitting operation and maintained in separate arrays. The edges and corners can then be iterated over directly if these filters are selected by the user. The match factor at a location is computed by iterating over every pixel inside the boundary of the pattern positioned at that location, as shown in the following equation:

$$MF = \sum_{i=0}^{ysize-1} \sum_{j=0}^{xsize-1} (pattern[i][j] * layout\_matrix[i][j])$$

Each complex number pixel in the layout is multiplied by the corresponding complex number pixel in the pattern, and every pixel's contribution is summed over the entire area.

Assume each complex number in the layout and pattern is purely real. If the layout pixel and pattern pixel have the same sign, then they match, and a value equal to the product of the magnitudes (importance) of those pixels is added to the match factor, increasing the correlation value. If the signs differ, then the match factor is similarly decreased. Zero pixels are "don't cares." Therefore, the closer a collection of layout shapes at the match location resembles the pattern shape, the higher its match factor will be. An example of running the bitmap algorithm on a small pattern is shown in Figure 16a.

The bitmap algorithm is simple but inefficient, and can only be run on small layouts. Speedup techniques such as image compression, layout partitioning, match location pre-filtering, and conditional code execution reduce the runtime significantly, but the computational complexity remains high. The runtime is proportional to the number of layout pixels, or the area of the layout. Large layouts must be partitioned into a very large number of areas so that the layout bitmap fits into main memory. Fortunately, there are a number of speedup methods based on algorithmic improvements that drastically improve the performance of the pattern matching process. These will be discussed in the following subsections.

There are a number of possible improvements to the bitmap algorithm that have not been tried, though it is doubtful that the bitmap algorithm with the product of these improvements can outperform the better basic algorithms explained in the next sections. The bitmap algorithm may be terminated earlier by matching on the highest weighted pixels first and skipping a location if the best 20-50% of the pixels leads to a poor match, and by skipping the contributions of pixels with a weight of nearly zero. Pixel-based learning can even be used to find the "important" pixels in the pattern. The coherency-based filtering method of the rectangle algorithm might be applicable to the bitmap algorithm as well. A random sampling of match locations over various places in the layout can quickly determine the proper match factor early termination threshold, coherency error bound, and match factor cutoff. An edge skip factor (ESF) can be used to only compute the match factor at every Nth pixel and then interpolate the match factors in between if they are expected to be high.

In the following sections, the primitive is assumed to be completely inside the pattern area. If the primitive extends beyond the pattern boundaries, then it must be clipped. Clipping is explained along with the triangle algorithm in Section 6.5, since this operation is simple for edges and rectangles but nontrivial for triangles.

6.2.1 Compression

The runtime of the bitmap algorithm scales as the number of match locations times the area of the pattern in pixels, since the match factor depends on the contributions from each pixel. However, a significant amount of work can be avoided by approximating the match factor using a subset of the pixel values and filtering out low matches where the predicted match factor plus the maximum error bound does not exceed the cutoff threshold. One way of reducing the number of pixels that needs to be tested is by terminating the match factor computation prematurely if the match factor is low enough. This only provides a minor speedup since, on average, maybe 75% of the pixels must be considered before the match factor computation can be terminated.

A much better system for early filtering of low match factor locations is by using an adaptive grid approximation, a form of lossy image compression. The layout and pattern are each recursively "compressed" to yield approximating matrices of increasingly smaller size and lower resolution. The match coordinates, match requirements, and other coordinate data is similarly compressed by dividing the X and Y coordinates by two. Each coordinate is flagged so that the set of original coordinates that were compressed to produce that value is known. At each level of compression, each group of two-by-two pixels in the layout and pattern is replaced by a single pixel with weight equal to the average of the original four. Thus, at each compression step, the matrix is reduced by a factor of two in each dimension for an overall reduction of a factor of four. If the match factor is computed at every point in the layout, then the number of locations to be matched on and the time per match factor computation are both reduced by a factor of four, giving a theoretical maximum speedup of 16 over the next higher resolution matching phase.

This speedup assumes that nearly all match locations are filtered out as poor potential matches based on the match factors estimated from the compressed pattern match. A threshold match factor cutoff value is determined by estimating the worst-case error in the match factor resulting from the pixel averaging due to compression. This error is related to the ΔMF pattern smoothness values described in Section 7.5.4.

The maximum 4X compression error is computed as follows. Given a cell matrix block of four pixels with values a, b, c, and d that are to be compressed into a single pixel of value e, the compression function is a simple averaging of the pixels. Similarly, a block of four pattern pixels with values f, g, h, and i are compressed into a single pixel with value j, resulting in a compressed multiplication product of m:

$$e = (a + b + c + d)/4$$

$$j = (f + g + h + i)/4$$

$$m = ej = (a + b + c + d)\ (f + g + h + i)/16$$

$$= (af+ag+ah+ai+bf+bg+bh+bi+cf+cg+ch+ci+df+dg+dh+di)/16$$

The exact multiplication product assuming no compression is computed as:

$$M = af + bg + ch + di$$

The compression error E is defined as the difference between the exact and compressed products:

$$E = M - m\ = M - ej =\ (af + bg + ch + di) - (a + b + c + d)(f + g + h + i)/16$$

The maximum error can be determined by choosing the values of a through d and f through i as either the most positive or most negative layer weight so as to maximize the difference between M and m. Two solutions exist: maximize m and force the product of e and j to be as negative as possible (max positive error E^+), or minimize M and force the product of e and j to be as positive as possible (max negative error E^-). The error estimation algorithm assigns maximum layer weights to the variables and chooses an error margin equal to $E^+ - E^-$. The relative error, $(E^+ - E^-)/M_{expected}$, is the parameter that determines cutoff values and limits the usefulness of compression. The relative error is specific to the layout and pattern values and is usually very conservative, so it can often be divided by a factor of two or so with a low probability of missed matches or errors in the results.

The matching run proceeds as follows. First, the layout and pattern matrices are compressed several times. The matching algorithm is run at the lowest resolution (smallest matrix sizes), and all locations with a match factor below the threshold are filtered out. The remaining set of locations is much smaller than the original set of all layout locations, though some of the remaining locations can represent up to four locations at the next highest resolution. Next, the matching algorithm is run on the next higher resolution representations of the layout and the patterns, but this time the match factor is only computed at locations that passed the previous filtering step. This process continues until the highest resolution level (no compression) is reached, at which point there are relatively few match locations left to consider. The final matching run is thus very fast and should still return the top match locations.

The choice of maximum compression level and filtering threshold determines the speed vs. accuracy tradeoff of the compression algorithm. Conservative values lead to a 2-4X runtime speedup due to compression with zero error, while more aggressive compression parameters can lead to a 10X speedup, though there is a small chance that some of the results may be missed. Experimental results showed that a single level of compression reduces the runtime by a factor of two to four, two levels of compression by a factor of up to seven, and three levels of compression by a factor of up to ten. Further levels of compression do not provide a significant speedup due to compression overhead. One level of compression rarely results in missed matches, while more than one level of compression can result in errors if the compression adjustment and correlation factors are set incorrectly. The optimal compression parameters depend on the layout and pattern statistics and cannot easily be determined, so choosing the compression parameters can be a trial-and-error process.

6.3 *Edge Algorithm*

6.3.1 1D Pre-Integration

One way to reduce the match factor computation time is to avoid touching each of the many pixels in the pattern. Since the sum of pattern pixel values overlapping a polygon is required for the match factor, and iterating through each pixel is not desired, the pattern values must be rearranged so that the sum of a large number of pixels can be determined with only a few operations. In the extreme example where the pattern is entirely covered by a single shape, every pixel value in the pattern must be summed to determine the weight of the pixels under the shape. Obviously, it is much better to sum up the pixels only once and then use that single number every time this situation occurs. Similarly, the sum of the right half of the pattern can be pre-computed, stored, and used any time a shape overlaps exactly the right half of the pattern.

It may appear that there are an enormous number of pixel group combinations that must be stored in order to make this algorithm work. However, consider the case where a shape exactly overlaps the left half of the pattern. The sum of the pixels overlapping this area can be pre-computed and stored, or it can be calculated by subtracting the sum of pixel values in the right side of the pattern from the total sum of pixels in the entire pattern. In this way, a series of additions and subtractions of a small number of carefully chosen pixel blocks will yield the area covered by a huge combination of actual pixel blocks. This is the basis of an algorithm that pre-integrates the pattern in various directions and allows a series of differences of pre-integrated table lookups to determine the sum of pixel values under a shape of arbitrary size and location.

In order to implement this idea and efficiently add contributions from edges or rectangle strips to the match factor, the pattern can be pre-integrated either horizontally or vertically in one dimension (1D). The following discussion assumes horizontal edge strips and thus horizontal pre-integration directed to the right, denoted by matrix IR. Each Y row of the pattern value matrix (PV) = PV[0,0] to PV[Nx-1,Ny-1] is processed in sequence, and a row of a pre-integrated matrix IR is computed in a way such that IR[X,Y] = sum from {i = X to Nx-1} of PV[i,Y], or IR[X,Y] = IR[X+1,Y] + PV[X,Y]. Each pre-integrated matrix element in IR is equal to the sum of all elements in PV at that location and to the right, as in the following equation:

$$IR(i,j) = \sum_{k=i}^{X} PV(k,j)$$

Then, if a rectangle strip spans from location (X1,Y) to (X2,Y), instead of summing each pixel from X1 to X2 in row Y using PV, the same number can be obtained from the difference of pre-integration matrix values: IR[X1,Y] – IR[X2,Y]. The number of operations is thus reduced from (X2-X1) to 2. This idea is used in the edge algorithm, which is described in the following section.

6.3.2 Edge Algorithm Details

A useful observation can be made about the layout pixel values. Since the layout consists of only a few unique layers, for instance 0- and 180-degree phased mask regions, the layout pixels take on a small number of discrete values. Moreover, these values appear together in large blocks of pixels because the grid resolution is usually much smaller than a feature size. This is especially true for large rectangle fills. Similarly, large areas of zero pixels, which denote the absence of all layers, are also common and can be skipped completely.

Each polygon in the input layout is first split into rectangles, and the rectangles are further split into either horizontal or vertical strips one grid unit in width that extend from one side of the rectangle to the other. The strips are then split into segments of length one grid unit, sorted, and stored in an array for each grid scan line of the layout. The intersections of multiple edges on different layers are summed up into a single intersection having a weight equal to the sum of all edge layer weights. The intersections along a row or column can then be queried by using a binary search and can be iterated over.

The following example demonstrates how this observation can be used to speed up the match factor computation. Assume that a large rectangle N pixels in width overlaps the pattern and extends from pattern column X to column X+N. Take a row Y in the pattern that lies between the top and bottom of the rectangle. If the pattern is pre-integrated to the right as in Section 6.3.1, then the sum of the pattern values PV[X,Y] to PV[X+N,Y] can be computed with two operations: IR[X,Y] – IR[X+N,Y]. This pre-integrated value is then multiplied by the weight of the rectangle to yield the match factor contribution from that horizontal strip of the rectangle. The 1D pre-integration matrix can be used to compute the sum of pattern values under a rectangle strip (between two edges at a particular Y value) in only two operations instead of N. The contribution of the entire rectangle to the match factor is thus the sum of the contributions of each row Y contained in the rectangle. Figure 10 demonstrates how the edge intersection algorithm works, and Figure 16b provides a numerical example of the edge intersection algorithm on a small pattern.

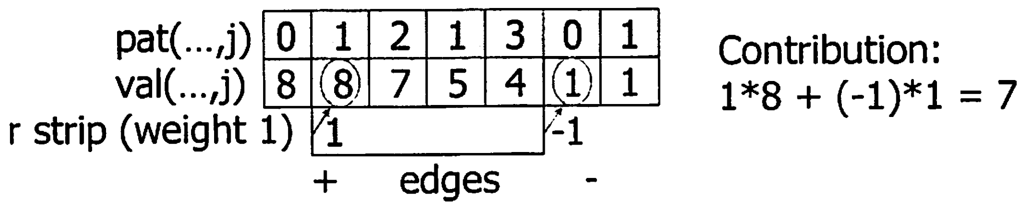


Figure 10: Demonstration of edge intersection algorithm: pat() is the original pattern matrix, val() is the pre-integrated matrix, and one horizontal rectangle strip is shown. Four bitmap operations have been replaced with two pre-integration matrix lookups.

The image compression algorithm as described in Section 6.2.1 can also be used with the edge algorithm, but the theoretical maximum speedup per compression level is only 4 as opposed to 16 for the bitmap algorithm. This is because the layout and patterns can only be compressed in one of the dimensions, the dimension along the pre-integration direction. In practice, compression of edges is not very helpful due to the increased overhead.

The runtime of the edge algorithm is proportional to the total perimeter of the layout shapes, which is a great improvement over the bitmap algorithm. There is an additional log term resulting from the binary search time to find the initial edge intersection with the right side of the pattern, but this term is usually insignificant. In practice, the edge algorithm is an order of magnitude faster than the bitmap algorithm, especially on multilayer layouts. In addition, the memory requirements are greatly reduced since only the edge pixels must be stored in the X- and Y-intersection lists. When using the edge algorithm, layouts as large as several square millimeters can be processed in only minutes.

6.4 *Rectangle Algorithm*

6.4.1 2D Pre-Integration

The previous sections explained how the pattern could be pre-integrated in 1D. This idea can be extended to pre-integration in 2D so that rectangular blocks of pixel values are stored instead of only 1D strips, as explained in the following section. A 2D matrix (P0) pre-integrated to the right and above is obtained by taking IR and performing the 1D pre-integration in the Y-direction so that P0[X,Y] = sum from {i = Y to Ny-1} of IR[X,i], or P0[X,Y] = P0[X,Y+1] + IR[X,Y]. Each matrix element in P0 represents the sum of pattern values PV at that location and in the pattern area above and to the right of that location, as shown in the following equation:

$$P0(i,j)=\sum_{k=i}^{Y}\sum_{l=j}^{X} PV(k,l)$$

A graphical diagram and example of 1D and 2D pre-integration is depicted in Figure 11.

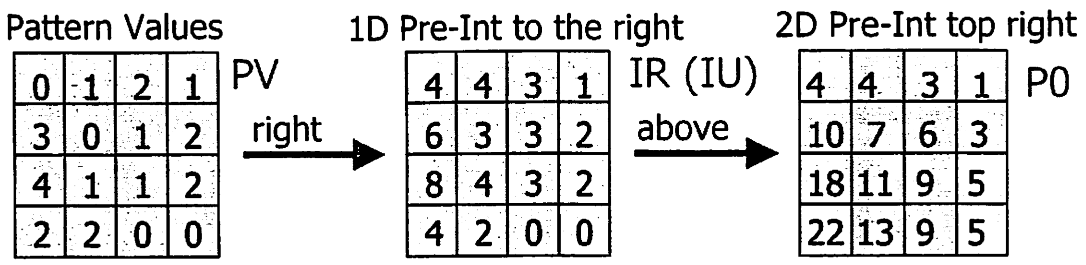


Figure 11: Illustration of 1D and 2D pre-integration

6.4.2 Rectangle Algorithm Details

The edge intersection algorithm can be taken a step further for even more efficient addition of rectangle contributions to the match factor. If the pattern can be pre-integrated in 1D to allow 1D rectangle strips to be added efficiently, then the pattern can be pre-integrated again in an orthogonal direction to allow an entire 2D rectangle to be added even more efficiently. This was demonstrated in the previous section.

The following text extends the example of Section 6.3.2 using the 2D pre-integration method developed in Section 6.4.1. Refer to Figure 12 for a graphical version of the steps involved in this example. Assume an N by M rectangle extends from lower left corner X,Y to upper right corner X+N,Y+M. The sum of pattern pixel values under the rectangle can be computed starting with the sum of all pattern pixels above and to the right of the lower left corner (X,Y) of the rectangle. This may overestimate the size of the rectangle. The problem is corrected by subtracting off the sum of pattern pixels above and to the right of the upper left corner (X,Y+M), and subtracting the sum of pattern pixels above and to the right of the lower right corner (X+N,Y). Since some area has been subtracted twice, it must be corrected by adding the sum of pattern pixels above and to the right of the upper right corner (X+N,Y+M) of the rectangle. In summary, the sum of pattern pixel values under the rectangle is equal to P0[X,Y] – P0[X,Y+M] – P0[X+N,Y] + P0[X+N,Y+M], where P0 is the 2D pre-integration matrix discussed previously. This number is then multiplied by the rectangle's layer weight to get the contribution of the entire rectangle to the match factor. This takes only four operations as opposed to N*M for the bitmap algorithm and 2*M for the edge algorithm, assuming the rectangles have been spatially sorted so that it is easy to determine which overlap the pattern at a given location. A numerical example of the rectangle algorithm is also given in Figure 16c.

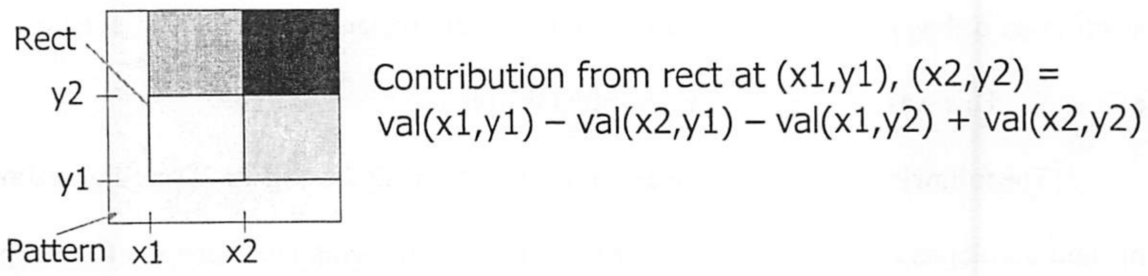

Rect
y2
y1
Pattern
x1
x2
Contribution from rect at (x1,y1), (x2,y2) =
val(x1,y1) – val(x2,y1) – val(x1,y2) + val(x2,y2)

Figure 12: Illustration of rectangle algorithm

The rectangle algorithm's data structure only contains information about the rectangles in a certain area of the geometry. The rectangles are flagged with information from the polygon splitting process regarding which edges and corners are true polygon edges and corners. The polygons overlapping a given partition can be easily and quickly extracted directly from the hierarchy without converting the geometry into a bitmap. Since the hierarchy is no longer flattened, the algorithm can run on any layout that will fit in memory in compact hierarchical form. The runtime of the rectangle-based algorithm is equal to C_1*M*N, where N is the number of match locations of interest, M is the match factor computation time per point, and C_1 is a constant based on the computer's architecture. M is linear in the number of rectangles overlapping the pattern (4*r), which can be determined for each location in the layout in time linear in the number of total rectangles R. The average number of rectangles overlapping a pattern is effectively the density of the layout, D = R/area. The number of edge/line end match locations is approximately proportional to R assuming the edges are tested at regular intervals on the layout grid. Therefore, the total runtime of the matching algorithm is equal to C_2*R*D, which overall grows as O(R) for a fixed technology design where density D is a constant.

Extracting a local set of rectangles from the hierarchy can take time O(n*log(n)) since the number of levels of hierarchy roughly grows as log(n), where n is the number of rectangles in the hierarchical layout description. The time taken to sort the rectangles is R*log(R/#partitions), where R/#partitions is related to the average number of rectangles in a partition. If the partition size is kept constant and all rectangles of the working set can be stored in memory, then log(R/partitions) is also constant and the rectangle sort time is O(R). In practice, the runtime is dominated by the matching loop due to the large value of constant C_2. The inner matching loop typically consumes 60-80% of the total runtime.

The only disadvantage to using rectangles is that it is difficult to determine which parts of each rectangle's sides are real edges or line ends and which corners appear on the periphery of the layout shapes. Approximately 10-20% of the overall runtime can be taken to compute the set of real edges and corners.

The rectangle algorithm does not benefit from layout and pattern compression since it does not depend on the actual number of pixels in the layout and the patterns. However, the rectangle algorithm can still be made faster by reducing the number of match locations. For example, instead of testing every pixel along a feature edge, every fourth pixel can be tested. In fact, if the pattern values are smooth (slowly varying), then the match factor at these intermediate pixels can be interpolated from surrounding points, and if necessary the exact match factor can be computed at these pixels if the interpolated value is above a certain threshold. This provides yet another way of filtering out poor match locations in order to reduce the runtime.

The rectangle algorithm is by far the fastest and most efficient of the three, having an order of magnitude speedup over the edge algorithm. The runtime is proportional to the number of shapes in the layout and the density of shapes in the layout, and no longer depends on the grid resolution. Thus, a higher grid resolution can be used for reduced discretization error without fear of drastically increasing the runtime. The runtime is, however, dependent on the technology since the density of shapes within the coherence radius of most processing effects increases as the k1 value increases. With the rectangle algorithm, layouts as large as several square centimeters can be processed in less than an hour. Similar algorithms are used for geometry processing in OPC calculations, such as the polygon convolution algorithm of Cobb [10], which relies on lookup tables to store the linear intensity contributions from each polygon edge.

6.4.3 Sorting

Input layout primitives such as paths, polygons, and circles are split into rectangles, where non-orthogonal edges are split on a fine grid or converted to triangles as shown in the next section. Arbitrary shapes can be processed, though diagonal lines can lead to many rectangles.

After subdividing the layout into small regions of rectangles equal in area to the largest pattern, at most four (two-by-two) regions overlap the pattern and thus it is easy to determine the rectangles whose weights contribute to the match factor at a collection of points. The lower left corner of the pattern is used to determine in which two-by-two region, or super-region, the pattern lies. A new data structure is built that contains the list of unique rectangles in each two-by-two combination of regions, and the match factor computation inner loop finds the bin that contains the pattern and iterates through the unique rectangles in that two-by-two block of regions overlapping the pattern. The rectangle pointers within each two-by-two bin are sorted in increasing order by y1, y2, x1, and then x2 in sequence. Thus, on average at least 25% of the rectangles iterated over actually overlap the pattern (75%

misses), and with early termination of the iteration when the first rectangle is found to be completely above the pattern, this "hit rate" can be increased to 33% (67% misses). More complex sorting methods and data structures can be used to increase the ratio of "hits" to "misses", though the increased preprocessing time and memory requirements generally make this not worthwhile.

The rectangles are initially sorted by region and are therefore iterated by region in the match factor computation loop in order to minimize cache misses. It is also easy to perform overlap removal, intersection tests, and layer Booleans on sets of spatially subdivided and sorted rectangles. The regions are iterated over, and each shaped in a region is tested against all other shapes in the region.

6.4.4 Inner Matching Loop

The inner matching loop typically consumes around 60% of the overall runtime of the pattern matcher. This simple loop is shown in Listing 2. The time is divided between checking for overlaps between rectangles and the pattern bounding box, accessing memory in the pattern pre-integration array P0, and performing the complex number multiply between the rectangle weight and the pattern pre-integrated result.

<u>Listing 2:</u>

```
Determine 2x2 region (super-region) index I in which pattern P lies
For each rectangle R in super-region[I] {
      If (R overlaps P) {
            Clip R by the bounding box of P; R = {x1, y1, x2, y2, layer}
            MF+=Weight(layer)*(P0[x1,y1]-P0[x1,y2]-P0[x2,y1]+P0[x1,y2])
      }
}
```

The overlap check runtime has been reduced by sorting the rectangles by region for improved memory access patterns. Some memory operations are avoided by skipping the pre-integration matrix contribution of rectangle coordinates that are outside of the pattern. The pattern pre-integration table cache misses can be minimized by interleaving the real and imaginary parts of the complex numbers when each component is nonzero, and storing only the nonzero component separately when one component is always zero (purely real or purely imaginary patterns). Finally, the complex number multiply can be removed if all rectangles have the same weight, and the sum of all rectangle contributions can instead be multiplied by the rectangle weight once at the very end. If there is more than one layer with the same weight, then some of the four floating-point multiples involved in the complex number multiply can be removed in certain cases. For instance, if the pattern or layout contains either purely real or purely imaginary values, then some of the terms involved in the complex multiply will always be zero, and their computation can be skipped. The actual inner loop for the rectangle and triangle algorithm is thus several hundred lines of code due to the various case splits that are used to take advantage of these special situations.

6.5 Triangle Algorithm

Some input layouts contain diagonal edges, usually with angles at multiples of 45 degrees where the technology allows. For example, permitting polysilicon lines to jog diagonally around contacts can lead to denser layout structures. These non-Manhattan polygons must be split into a very large number of rectangles to accurately approximate the sloped edges. In many cases, right triangles can be used as an alternative to replace these large collections of tiny rectangles in efficiently representing a diagonal polygon edge. The number of primitives will often be reduced to a fraction of the size of the required rectangle set. However convenient this may seem, it comes at a price: triangles are much more complex to process and involve the computation and storage of a larger number of pre-integration matrices.

Triangles are represented as rectangles with vertices corresponding to the triangle's bounding box and an extra bit flag denoting which orientation the triangle is in. There are four possible orientations, one for each location of the right angle, and the rectangular bounding box and orientation flag uniquely identify a right triangle's vertices. Shape processing is nearly identical for rectangles and triangles, and most of the operations support triangles by checking for the orientation flags in special case triangle code. One exception is overlap removal, which is difficult to perform between rectangles and right triangles and is therefore not currently supported.

6.5.1 8-way Pre-Integration

In the case of the four possible 45-degree triangle orientations shown in Figure 13a, eight 2D pre-integration matrices are used, one for each quadrant (P0, P1, P2, P3) and one for each of the four lower octants (O1, O2, O3, O4). Triangles with angles that are not a multiple of 45 degrees are rarely encountered in integrated circuit layouts due to layout generation algorithm complexities and manufacturing difficulties. Supporting these triangles would require an extremely large set of pre-integration matrices, so they are not supported in the general triangle algorithm. Details on extensions to non 45-degree triangles are explained in Section 6.5.3. See Figure 13b for a diagram of the area (angular rotations) covered by these pre-integration matrices. There are a number of other possible combinations, including some using only six matrices with added computation steps, but they will not be discussed here. These matrices are pre-computed for 90- and 45-degree angular regions beginning at a point in the pattern and continuing to the edges of the pattern matrix. Pixels that lie exactly on a diagonal edge can be fully counted, not counted at all, or counted as one half their actual pattern values for highest accuracy.

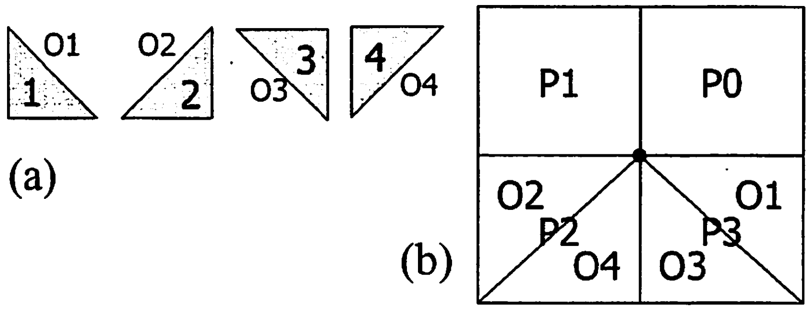


Figure 13: (a) Four possible triangle orientations and (b) 8-way triangle algorithm pre-integration area

Right triangles are used because they have fewer degrees of freedom and therefore do not require as large of a pre-integration matrix. Most integrated circuit layout polygons can be exactly divided into right triangles. Odd angles of less than 90 degrees where both incoming and outgoing edges lie within the same quadrant cannot be split into right triangles, so in those cases they are split into a number of approximating rectangles and triangles. There are four possible right triangle orientations corresponding to four unique positions of the right angle. The remainder of this section will assume the triangle is in the first orientation with its right angle in the lower left corner and its hypotenuse oriented from the upper left to the lower right. Two 2D pre-integration matrices are required for this case. Matrix P0 represents the sum of pattern pixel values above and to the right. Matrix O1 represents the sum of pixel values in the octant between 315 and 360 degrees.

Figure 14 explains the steps involved in using the triangle algorithm for triangles in this orientation, and an example is provided in Figure 16d. First, the P0 pre-integration matrix is used to retrieve the sum of pattern pixel values above and to the right of the triangle's right angle point at X1,Y1. Then P0 at the top left corner (X1,Y2) of the triangle is subtracted, leaving an area of pattern pixel values that extends from the left vertical leg of the triangle to the right side of the pattern area and includes the entire triangle. Next, the value of O1 of the top left (X1,Y2) triangle point is subtracted to remove the pattern pixel values to the right of the triangle's diagonal edge. This operation subtracts a 45-degree angular slice of the pattern between 315 and 360 degrees. If this area extends beyond the bottom of the triangle, the extra area must be added back in and is equal to the value of O1 at the lower right (X2,Y1) corner of the triangle. Thus four operations are involved: P0(X1,Y1) – P0(X1,Y2) - O1(X1,Y2) + O1(X2,Y1).

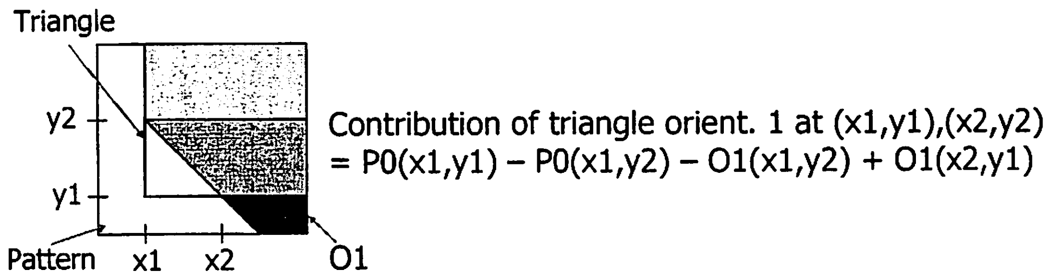

Triangle
y2
y1
Pattern
x1
x2
O1
Contribution of triangle orient. 1 at (x1,y1),(x2,y2)
= P0(x1,y1) – P0(x1,y2) – O1(x1,y2) + O1(x2,y1)

Figure 14: Illustration of 45-degree triangle algorithm

A triangle may need to be split if it only partially overlaps the pattern, and the memory lookup is in a large set of pre-integration matrices. In addition, contributions from triangles that are not multiples of 45 degrees must be added using a variant of the edge algorithm. Thus, triangle processing is somewhat slower than rectangle processing.

6.5.2 Clipping

One problem with the matching algorithms as presented so far occurs when the primitive shape only partially overlaps the pattern area. If the shape extends beyond the pattern area, then lookups in the pre-integration tables outside of the computed region will fail. This means that the shapes must be clipped to the pattern area before the table lookups are performed. In the case of edges and rectangles, these primitives are easily clipped to the bounding box of the pattern using min() and max() to yield a primitive of the same type that lies completely within the pattern.

However, additional difficulties arise when clipping a triangle to a rectangular area. Consider the triangle clipped to the pattern bounding box in Figure 15b. The intersection of the two shapes is no longer a triangle; it is a five-sided polygon. This shape must be further split on the fly into smaller shapes consisting of a possible smaller triangle of the same orientation and up to two new rectangles. There are a number of cases to consider when splitting the polygon resulting from clipping, but fortunately these cases can be enumerated efficiently without having to resort to the slow arbitrary non-Manhattan polygon splitting algorithm explained later in Section 7.4.1. The contributions from each of the resulting primitives are then added to the match factor in the usual way for rectangles and triangles as discussed in Section 6.4 and Section 6.5. The worst-case number of operations required to add the contribution of a 45-degree right triangle to the match factor is therefore actually 12, since each of up to three shapes requires four operations.

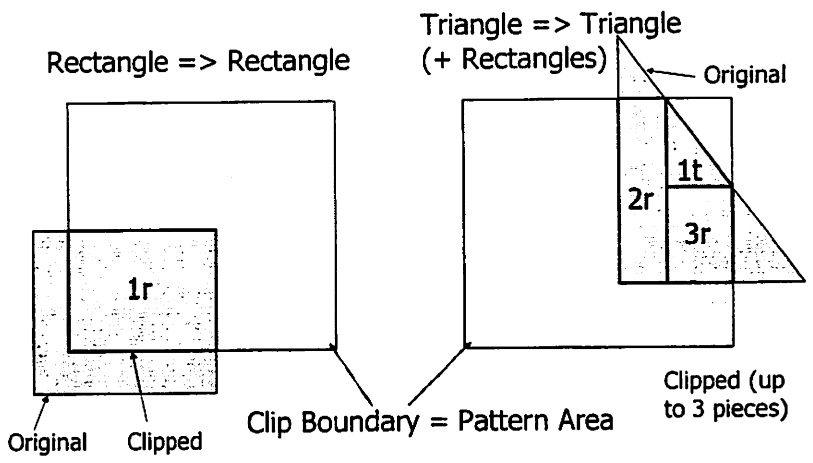


Figure 15: (a) Rectangle and (b) triangle clipping

6.5.3 Extensions for non 45 Triangles

The above triangle algorithm is capable of processing triangles with angles at multiples of 45 degrees. A problem arises when a polygon contains an angle that is not a multiple of 45 degrees. These polygons can occur, for example, in a spiral inductor of an analog integrated circuit layout. In fact, even a 45-degree polygon can be split into non 45-degree right triangles if the original polygon does not lie on the layout grid used for triangulation and the vertices have to be snapped to the grid. Grid rounding error can result in a one to two pixel shift in edge and corner locations, thus perturbing the angle. The triangle algorithm from the previous section cannot be used in these situations.

These triangles can still be processed more efficiently than if they were split into many small, single pixel-width rectangles. One method to do this is to use a combination of both the rectangle algorithm and the edge intersection algorithm. The contribution of the rectangle representing the bounding box of the triangle extended to the vertical edges of the pattern opposite the triangle's hypotenuse is first computed. This area includes pixels that are not in the triangle. To correct for this, the pixels between the hypotenuse of the triangle and the opposite vertical edge of the pattern boundary can be subtracted in horizontal strips. This can be done using the pre-integration matrix from the edge algorithm (IR, integrated right, or IL, integrated left) in Section 6.3.1. The difference between this step and the edge algorithm is that the strip runs to the vertical edge of the pattern, and thus the pre-integrated value at the end of the edge is zero and that computation can be omitted. If the height of the triangle is H, then the number of operations performed is equal to H+2. Figure 16e shows an example of how this algorithm is applied to a small pattern.

The improvement in speed is not as impressive as in the 45-degree triangle algorithm. If rectangles are used instead of triangles then the number of operations is 4*H, and the number of operations is 2*H for the edge intersection algorithm. The real advantage of using this approach is that it greatly reduces the number of primitives, which in turn reduces the computation time for pre-processing the data and the memory required to store the primitives.

Figure 16 demonstrates the various algorithms from Section 6.5 with examples using a simple pattern and a single triangle primitive.

The pre-integrated pattern matching idea can actually be applied to any shape or composition of shapes satisfying the following requirements:

(a) A compact pre-integration matrix can be built for the shape at any location, size, and orientation,
(b) The shape as clipped by the pattern boundaries can be subdivided into shapes of a supported type, and
(c) The shapes can be combined into a rectangle.

Note that this includes edges, rectangles, and right triangles but not shapes such as circles or non-right triangles because they violate condition (c). It might be possible to relax condition (c) by using a non-rectangular pre-integration matrix, but this idea has not been verified.

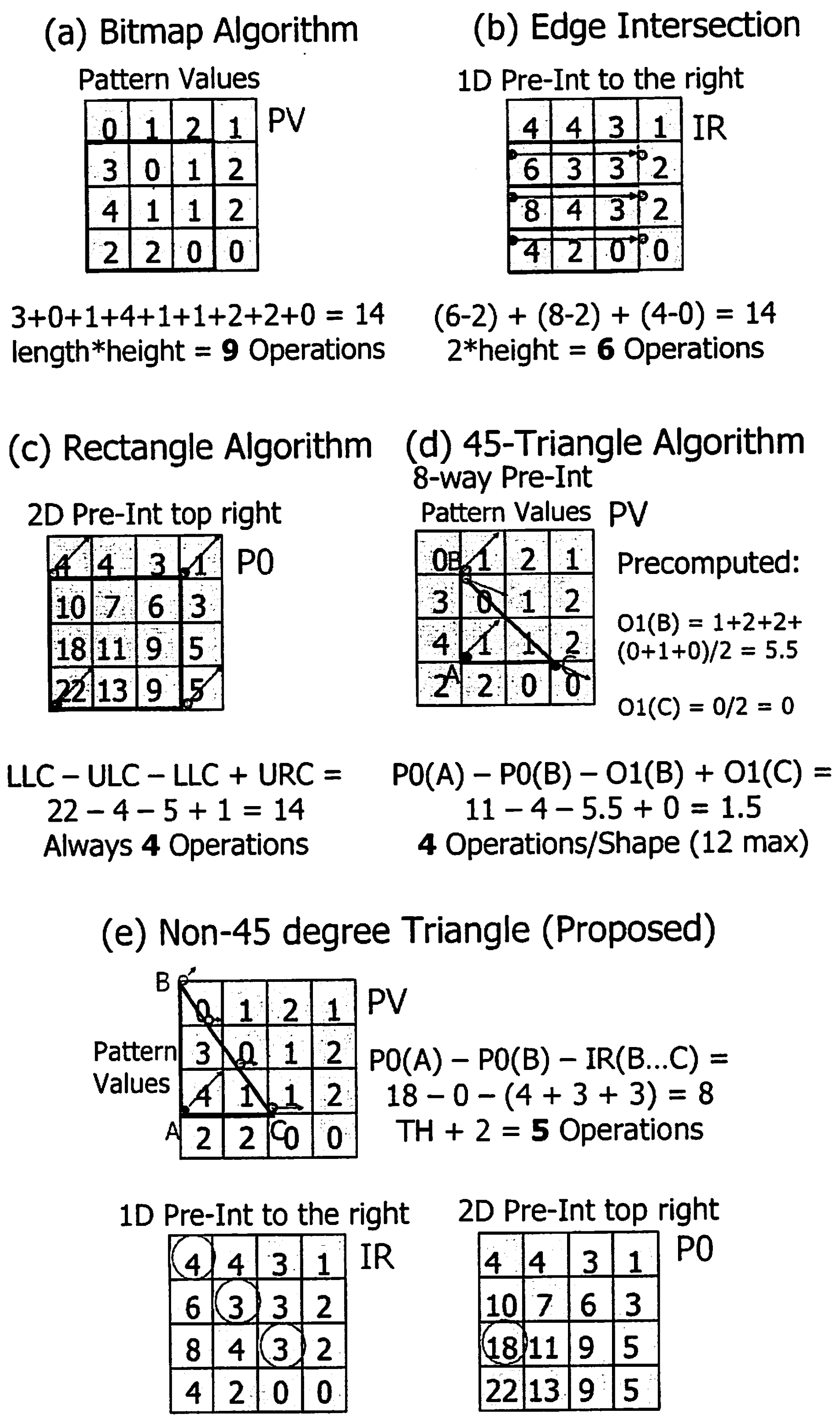


Figure 16: Example of adding the contribution of a rectangle and triangle to the match factor using the (a) bitmap, (b) edge, (c) rectangle, (d) 45-degree triangle, and (e) non-45-degree triangle algorithms

6.6 Other Algorithms

"Pattern matching" is a very broad term that encompasses matching geometry to geometry, images to geometry, and images to images. The type of pattern matching discussed in this dissertation is more of a search procedure for finding an image in a collection of geometric shapes, where the match factor is determined through an image correlation with the layout geometry. Thus it is important to also consider image convolution and correlation algorithms when discussing pattern-matching strategies. The basic differences between lithographic processing effect pattern matching and standard image pattern matching are really the larger size of the area in which the pattern is matched, and the ability to perform application-specific filtering of the match point candidates. Any class of pattern matching algorithm can in fact be used for applications such as lens aberration sensitivity analysis, since an image can always be converted into pixel-sized rectangles and the layout geometry can always be converted into a large bitmap image. However, one unique advantage of the pattern matching system described here is the ability to determine inexact matches, which cannot be easily done with many of the well-known matching algorithms.

Geometric pattern matching, or geometry-to-geometry matching, is a common method of locating unique polygons in a dataset for applications such as geometric data compression [38]. This class of algorithms is extremely efficient in locating all exact polygon shapes in a large search area. In fact, the pattern matching software system presented in Chapter 4 includes this type of geometric pattern matching algorithm for polygon data compression of the input layout, as discussed in Section 4.5.2.2. Unfortunately, this class of algorithms is not applicable to searching for the best match to smooth images representing processing effects.

Pattern matching is a well-known problem in the areas of digital image processing and computer vision, and there are a large number of image-to-image pattern matching algorithms in existence. These algorithms can be grouped as correlation-based solutions and image understanding solutions [45]. There are a number of differences between pattern matching over large integrated circuit layouts and pattern matching over standard images. The layout pattern matching discussed in this dissertation is made easier by the fact that there are a small number of discrete pixel values and large groups of the same pixel value. Layout pattern matching does not involve image scaling or rotation by arbitrary angles since the pattern size and orientation are generally fixed. In addition, the match factor need not be computed at every point in the layout image, so aggressive location filtering can be used. Most of the pattern matching algorithms discussed below do not take advantage of these optimizations.

Layout pattern matching is also more difficult than standard image pattern matching for several reasons. The layout "image" is orders of magnitude larger than what normally is considered an "image", and cannot be stored in memory at one time. These layout images are weighted by complex-valued floating-point numbers instead of the normal integer RGB image intensity values. This leads to more complex normalization of the correlation factor. Pattern matching on layouts also involves geometry processing to extract features and feature-dependent weighting of the results.

6.6.1 FFT-Based Algorithms

Correlation is algorithmically equivalent to convolution, so the inner pattern-matching loop is performing operation similar to convolving the pattern with the layout. Consequently, fast convolution methods such as derivates of Fast Fourier Transform (FFT)-based algorithms can be used for computing the match factor [44].

Assume, for simplicity, that the layout and pattern are of the same size. This assumption is valid because the layout can always be divided into smaller partitions that are the same size as the pattern and matched independently. A standard convolution in the context of the pattern matcher can be expressed as $MF = L \otimes P$, where MF is the match factor, L is the layout matrix, and P is the pattern matrix, all complex-valued floating point numbers. The computational complexity of basic convolution is $O(n^2)$, where n is the number of pixels in the layout and the pattern. The runtime of the bitmap algorithm on every layout pixel is equal to the number of match locations (n) times the number of pattern pixels (n), which is indeed n^2. A convolution can be computed more efficiently by utilizing the FFT and inverse FFT (IFFT) operations: $MF = L \otimes P = IFFT\{FFT\{L\} * FFT\{P\}\}$. The computational complexity (CC) of this method is equal to CC(CC(*, per-pixel multiply) + 2*CC(FFT) + CC(IFFT)) = CC(n + 2*n*log(n) + n*log(n)) = n*log(n), which is much better than a computational complexity of n^2.

This FFT-based algorithm achieves a speedup of n/log(n) over the bitmap algorithm. In addition, its runtime does not strongly depend on the pattern size. However, this algorithm does not take advantage of the fact that layout pixel values take on only a small number of values and are clustered into large groups of the same pixel values. Moreover, this algorithm always calculates the match factor at every point in the layout, even if only a small subset of the points along feature edges or corners is needed. Computing the FFT result at only a few pixels out of the total matrix appears to have the same computational complexity as computing the FFT values for all pixels. Finally, this algorithm requires additional memory to store the intermediate FFT matrices. Unless the pattern is very large or the match factor must be computed at a large percentage of the total layout pixels, the rectangle algorithm is still much faster than the FFT-based algorithm. In fact, the time taken to simply construct the matrices on which the FFT operations will later be performed is greater than the time taken to run the entire rectangle algorithm for several layouts that were tested. Though the FFT-based algorithm is interesting from a theoretical perspective, it is inefficient for practical matching runs on large layouts when compared to the efficiency of the rectangle and triangle algorithms. It may be possible to compute the FT of the layout very quickly by summing the individual Fourier Transforms of the rectangles in the layout using a table lookup, but this idea has not been tested.

6.6.2 Statistical Sampling Algorithms

Statistical pattern matching algorithms generally involve finding a pattern image located within a search image by randomly checking a pixel in the pattern against a pixel in the search area and throwing out a location if the pixel match fails. The set of test pixels is usually a constant size regardless of pattern size, and in some cases the pixels may be strategically chosen to improve the performance of the algorithm. This strategy can lead to extremely fast exact pattern matching due to the small number of initial pixels that need to be tested to filter out poor match locations. It can also handle certain types of rotations and scaling of the pattern. One paper on a statistical sampling algorithm [45] quoted a speedup factor of 160 over the basic bitmap algorithm.

This algorithm does not appear to work with inexact pattern matching since it reaches a decision after testing only a small fraction of the total pattern pixels. It is not clear whether the assumptions made in this class of algorithms hold for the case of aberration patterns matched on circuit layouts. Furthermore, these algorithms may not take advantage of the small number of discrete pixels values and the large blocks of the same pixel values present in circuit layouts.

6.6.3 Learning Algorithms

Learning algorithms involve training a system to accept good matches and reject poor matches through learning trials based on test images. The system is trained to locate better matches by processing a large number of patterns flagged as good and a similarly large number of patterns flagged as poor, and in the process building a method of differentiating the two cases. The training and matching can utilize neural networks or other decision processes. The pattern matching system can be trained to locate patterns that have been scaled and rotated, and the algorithm is robust to noise and low contrast levels. However, these features are not required for most of the layout pattern matching applications discussed in this dissertation.

The patented vsFIND algorithm [46] uses the highest "value" pixels to filter out poor matches very quickly, similar to but more powerful than bitmap pixel compression. The standard Normalized Grayscale Correlation (NGC) method is used to locate the pattern within the search image. The training time has been optimized to scale as the pattern width to the power of 3.5.

Learning-bases systems are typically used to determine if two equivalently sized images match and do not easily generalize to locating the target image in a large search image such as an integrated circuit layout. It is not clear how this type of algorithm would perform for finding aberration patterns in real layouts. The algorithms do not take advantage of the small number of discrete pixels values and the large blocks of the same pixel values present in circuit layouts. A learning algorithm would likely scale fairly well to large layouts, but the training process could require excessive runtime that scales very strongly with the pattern size. The training procedure might also be difficult to automate, especially for the smoothly varying aberration patterns that may have no critical image features to search for. Finally, in some cases the algorithm is not guaranteed to produce the correct results if using the most aggressive performance settings.

6.6.4 Rejection Algorithms (Inverted Pyramid)

This class of algorithms uses projection vectors to filter out non-matches very quickly and dramatically reduces the search space. In the case of the Projection Kernel algorithm [47], the match locations are repeatedly filtered using a Walsh-Hadamard tree, even in the presence of noise. The Maximal Rejection Classifier algorithm [48] iteratively classifies match locations into targets and clutter. The general rejection method starts out by performing the simplest tests first in order to filter out the obviously bad locations, which removes the majority of locations from consideration. These simple tests check the existence of important features of the pattern in the potential match location, such as key edges and corners. Increasingly more difficult tests are used as the potential match set is narrowed down to smaller and smaller sizes. The final set of results has passed all tests without being rejected, and therefore represents the target match.

These algorithms were not designed to locate inexact matches when the best match value is unknown, and poor matches cannot be removed initially because the best match factor is unknown until the algorithm is complete. These algorithms are much more efficient than the bitmap algorithm, but are unlikely to provide the orders of magnitude speedup of the rectangle algorithm. In addition, the Projection Kernel algorithm requires memory proportional to $P^2 * \log(P^2)$ as opposed to P^2 for the bitmap algorithm, where P is the pattern width and height (typically 128 or larger). This $2 * \log(P)$ increase in pattern memory requirements may potentially be prohibitive for large patterns.

6.6.5 Video and Image Compression Algorithms

Pattern matching methods have been proposed for high quality two-dimensional video and image compression in [49] and [50]. These algorithms allow for approximate matching of a region of an image to the recently compressed portion of the image or other previously processed images. This allows the software to locate pixel regions that can be copied from another location so as to avoid storing the similar data twice. This idea is similar to the lossless exact image pattern matching algorithm of Lempel-Ziv. Approximate pattern matching can also be used to find a previous video frame that is similar to the current frame in order to reduce the required data rate by copying pixel regions from one frame to another. These image compression techniques are interesting, but may not scale well for full chip layout-sized images and large patterns of several hundred pixels on a side.

6.7 Overview and Algorithm Computational Complexity

A flowchart showing the construction of several of the major data structures is shown in Figure 17. In all cases, polygons in the input file are first split into rectangles, since rectangles are easier to process than polygons. In the case of the triangle algorithm, the polygons are split into triangles as well as rectangles and the rest of the data structures and algorithms proceed in the same way as in the rectangle algorithm. The bitmap algorithm adds the rectangles to the Boolean layer map, where each bit of the layer map represents a different mask layer or phase. The individual layers of the layer map are then summed up into the complex-valued layout matrix, which is used for the pattern matching inner loop. Alternatively, the edge algorithm stores points along either the horizontal or the vertical edges of the rectangles. These edge intersections are stored individually for each row or column of the layout, sorted, and merged with other overlapping edges so that there are no duplicate points stored. This data structure is then used for edge-based pattern matching. As another alternative, the rectangle algorithm simply sorts the rectangles spatially into sub-regions, and maintains a pointer to each rectangle overlapping the sub-region. The set of rectangles in each two-by-two collection of sub-regions is also stored for use in the inner matching loop, where again duplicate overlapping rectangles are removed.

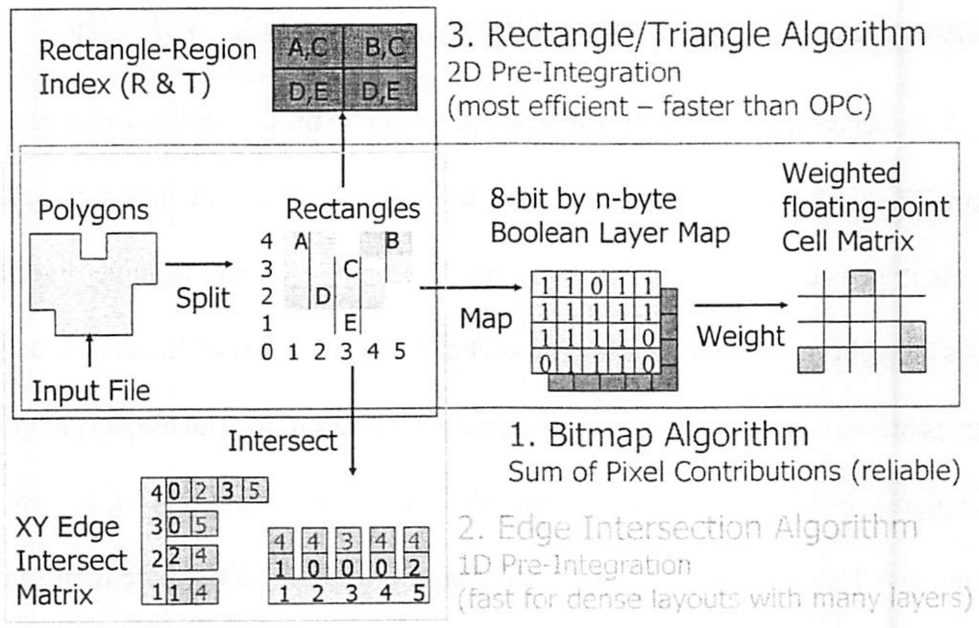

Rectangle-Region Index (R & T)
A,C
B,C
D,E
D,E
3. Rectangle/Triangle Algorithm
2D Pre-Integration
(most efficient – faster than OPC)
Polygons
Split
Input File
Rectangles
A
B
C
D
E
8-bit by n-byte Boolean Layer Map
Map
Weighted floating-point Cell Matrix
Weight
Intersect
1. Bitmap Algorithm
Sum of Pixel Contributions (reliable)
XY Edge Intersect Matrix
2. Edge Intersection Algorithm
1D Pre-Integration
(fast for dense layouts with many layers)

Figure 17: Pattern matcher data structures for the bitmap, edge intersection, and rectangle/triangle algorithms

The various matching parameters that affect runtime are described in Table 2. These parameters are for a standard benchmark matching run on the active_test layout. This layout was designed in 180nm technology, so the grid resolution was chosen to be 50nm so as to have 3-4 pixels per minimum feature. The coma pattern used for matching was 128 by 128 pixels in size, or 6.4μm on a side. The overall active_test layout is 2.4cm by 1.8cm, or 480Kpixels by 360Kpixels, yielding a total of 172.8Gpixels of effective matching area. Clearly it is not possible to create a bitmap this larger, nor is it efficient to generate one partition-by-partition. The number of actual edge match locations in this layout corresponds to the total perimeter of all shapes, which is 2.6 billion points, or 130m of total edge length! There are a total of 38 million flattened rectangles in a single layer, though the actual layout file is small due to the compression ratio of eleven levels of hierarchy. The optimal FFT size for this layout was determined to be 1920 by taking the derivative of the FFT computational complexity and solving for the value of X that makes the derivative equal to zero.

Table 3 presents an overview of the computational complexities of the various pattern-matching algorithms discussed in this dissertation. The algorithm computational complexities alone have many terms as explained in Table 2 and depend on non-standard input layout parameters. The estimated operation count for the active_test layout is a fairly good indicator of the actual algorithm performance on a real input layout. The operation counts were determined using the values of Table 2, but they do not directly determine runtime since different types of operations may take a different number of processor instruction cycles to execute. The "estimated true runtime" values were determined from experiment for the short runtimes of implemented algorithms, extrapolation of runtimes from smaller examples for the time-consuming implemented algorithms, and operation count figures for the unimplemented algorithms. This table proves that the edge algorithm is much faster than the bitmap algorithm for normal layouts, and the rectangle algorithm is much faster than the edge algorithm. The other algorithms are faster than the bitmap algorithm but slower than the rectangle algorithm, though they may be better for uncommon types of layouts. Additional data on algorithm performance can be found in Section 8.6.1.

Parameter	**Description**	**Value**
res	grid resolution	50nm
P	pattern size in (square pattern)	128 pixels
N	layout width	2.4cm = 480000 pixels
M	layout height	1.8cm = 360000 pixels
L	number of match locations along edges (perimeter)	2.6 billion
R	number of effective flat rectangles	38 million
np	number of patterns to match (multiplies all runtimes)	1
nl	number of layout layers	1
X	optional FFT partition size	1920

Table 2: Pattern matching parameters for algorithm computational complexity comparison

Algorithm	**Computational Complexity**	**Operation Count**	**Estimated True Runtime**
Generating the bitmap	M*N	1.72E11	2 hours[11]
Bitmap	$N*M*P^2$	2.8E15	> 1 year[12]
Bitmap (edges only)	$L* P^2$	4.3E13	1 month[12]
Bitmap (edges only + compression)	$C*L* P^2$	1.0E13	5 days[12]
Bitmap (edges only + compression + ESF)	$C*L* P^2/ESF$	4.0E12	> 1 day[12]
Edge	$L^2*P^2/(N*M)$	6.4E11	10 hours[13]
Rectangle/Triangle	$L*R* P^2/(N*M)$	9.4E9	34 minutes[11]
Fourier Transform (FFT)	$2*(X+P)^2*\log(X+P)*M*N/X^2$	4.4E12	2 days[13]
Projection Kernel	N*M*log(P)	1.2E12	-
Statistical Sampling	N*M*samples_per_pattern	-	160X speedup over bitmap[14]

Table 3: Pattern matching algorithm computational complexity comparison

11. Measured from system clock
12. Predicted from operation count
13. Extrapolated from smaller test cases
14. Estimated typical value taken from publication

7 Software Details

The pattern matching algorithms described in Chapter 6 in general only apply to the inner matching loop. The pattern matcher program contains a large variety of supporting features and algorithms that are mostly independent of the core matching algorithm. The following sections of Chapter 7 assume that the rectangle and triangle algorithm is in use for the case where there is a difference, since that algorithm has been chosen for the final pattern matcher implementation based on performance tests.

7.1 GDSII File Read

GDSII files can be read as ASCII text files, raw binary files, and zipped binary files, including multi-file archives. The file import code is highly optimized for reading large GDSII files of many GB using a dual pass read and on-the-fly data compression for reduced memory requirements of polygon and hierarchy storage. It is important to have a fast input file read because this segment of the runtime cannot otherwise be parallelized and in some cases could dominate the overall runtime.

GDSII files are read in two passes. The first pass computes the number of cells and number of shapes of each type in each cell, so that the entire database can be allocated at one time using large blocks of memory. This improves cache coherency and avoids the added runtime and memory required for dynamically allocated arrays. The bounding box of the layout is also estimated in the first pass, and is used to create the spatial subdivision bins so that the layout can be directly read into the sub-cells. The second pass then fills in the database, adding objects into the already allocated space within the target cell or sub-cell. This two-pass algorithm can build the database extremely quickly so that the load time is often dominated by disk I/O and unzipping of the input data, especially for flat cells.

The GDSII file loader includes a preprocess file mechanism for avoiding the first pass on repeat loads of the same layout. Normally, the first GDSII read pass is used to determine the number of cells, bounding box of the layout, and number of objects of each type per cell. This is a relatively small amount of information compared to the entire layout, and this information is dependent solely on the layout. Therefore, this first pass data can be computed the first time a layout is read and stored in a file. If the same layout is read again later, then the required information is loaded from the preprocess file instead of the layout, avoiding one pass through the GDSII file. The resulting load time is improved by as much as a factor of two. Other information can be stored in the preprocess file, including sub-cell data statistics and verification data such as a GDSII file name, timestamp, and/or checksum values.

The parallel UNIX version of the pattern matcher described in Section 7.8 also employs a pipelined disk read/unzip/database build in both read passes. In the first step, thread 1 reads block A of layout data, where the block size is experimentally determined to be in the range of several megabytes. Then in the next step, thread 1 reads block B of layout data into one memory buffer while thread 2 processes block A, which is stored in a second memory buffer. At every step, thread 1 reads block N+1 while thread 2 builds the database for block N, and the step ends with a thread synchronization event. Disk reads are generally supported in hardware, so the processor is free to do data processing while waiting for the read, and a speedup is possible even on a single processor machine. The disk read and unzip can then occur in parallel with the data processing, yielding a speedup of up to a factor of two.

Since the disk read is inherently sequential and the unzipping cannot easily be done in parallel using ZLIB, this is near the theoretical minimum load time. It may be possible to fully parallelize the zip file read by preprocessing the zip archive so that smaller groups of records are independently compressed. At load time, the processors can each read a subset of the compressed sections of the layout and in that way unzip the data and build the database in parallel. Then the individual processors can exchange their portion of the database through network communication, so as to spread the communication out over the entire network instead of constantly retrieving data from the server that contains the original zip file on disk.

7.2 *Hierarchy*

Many modern integrated circuits are designed using a hierarchy of standard cells, macrocells, and/or possibly underlying clock and power grids. Large arrays of regular structures such as on-chip RAM, cache, clock and power grids, and tiled gate arrays are poor candidates for pattern matching because the large number of repetitive structures leads to many matches with identical values, producing inflated sets of match results. These areas of the design are best tested by running the matching algorithm on a single tile of the array. However, more irregular cell hierarchies are well suited for pattern matching.

GDSII and CIF files allow an arbitrary amount of hierarchy in order to reduce the size of the geometry representation, and the pattern matcher preserves the hierarchy up to the partitioning stage to minimize memory requirements. The import-enabled pattern matcher and simpl_display share the same hierarchical database, display, and query routines as explained in Section 4.5.2. Initially, the imported layout is built into a hierarchy of cells, each of which contains polygonal shapes and references to other instantiated cells. The cell hierarchy is preserved throughout the execution of the pattern matcher and used for displaying the layout upon completion of the matching run. Each partition is iterated over, and the pattern matcher queries the hierarchical database to determine which cells and included shapes overlap the current partition. Those overlapping shapes are flattened and then split into geometric primitives for use in the later match factor computation step.

The pattern matcher's support of hierarchy has evolved over time and is still in an unfinished state. The original pattern matcher supported only flat layouts. Later, cif_to_pm was developed to allow hierarchical layouts to be read in and flattened for use in the core pattern matcher, as explained in Section 4.4. The next step was to store the hierarchy in memory and directly access the partition data from the hierarchical cell database and only flatten a single partition at a time. A challenging future step would be to directly match on the hierarchy without any flattening of the database, or to go even further by computing the match factors on common cells and replicating the computed value across all instances of that cell. There are a number of potential problems with this approach, such as determining which of the millions of points in the cell to compute and store the match factors for.

7.3 Partitioning

Large layouts of hundreds of millions of shapes cannot be flattened and stored in memory at one time. In order to process these layouts, they must be spatially subdivided into a number of smaller partitions that can be flattened and stored in memory at once. These large layouts can either be preprocessed and written to disk or constructed directly from the hierarchical database. If the hierarchical database is not in memory, then the layout is partitioned using an external bucket sort and stored on disk for later access to per-partition data. Polygons overlapping the edge of a partition are clipped by the partition boundary. The partitions are then iterated through, and only the flat data for the current partition is loaded and used in the matching loop. The graphical version of the pattern matcher also loads and displays each visible partition. Rectangle, triangle, and polygon data is retrieved from the hierarchy or the rectangle database in memory or on disk.

It is important to choose a proper partition size in order to minimize runtime and memory requirements. If the partition size is set too small, then there will be an extremely large number of partitions leading to long partitioning times, potential disk swap, and excessive memory required for the partition structure itself. If the partitions are too large then the flat geometry will not fit into cache, or even worse, not fit into main memory. This can result in a large ratio of cache misses to cache hits and possibly costly swapping to disk as some of the geometry is stored in virtual memory. It takes longer to spatially sort the data of large partitions as well, but if the partition size is kept small so that only the number of partitions increases with the size of the layout then the sort time is effectively linear in the number of shapes. The partition size is especially critical for the bitmap and edge algorithms, as they require a larger number of primitives and thus more memory to represent the layout. In addition, the partition size impacts the granularity of the parallel pattern matching algorithm described in Section 7.8. Fortunately, there is a large flat area in the runtime curve, so partition sizes within a wide range work reasonably well.

The layout can be partitioned into a large number of equally sized areas that can then be locally flattened and processed independently from other partitions. However, when the point of interest lies near the edge of the partition, the pattern may extend beyond the partition boundary. The geometry near the edge of the partition boundary but outside of the partition area must also be stored so that the contributions from these shapes can be added to the match factor. This requires the partitions to overlap with their left and right neighbors by an amount equal to half of the largest pattern width and with their top and bottom neighbors by an amount equal to half of the largest pattern height. This overlap effectively increases the partition size, so the partitions should be made much larger than the largest pattern size so as to minimize the overlapped area relative to the core partition area. The optimal partition size is usually much larger than the pattern anyway, so the partition overlap poses no real problem. A small layout that has been divided into six overlapping partitions is shown in Figure 18.

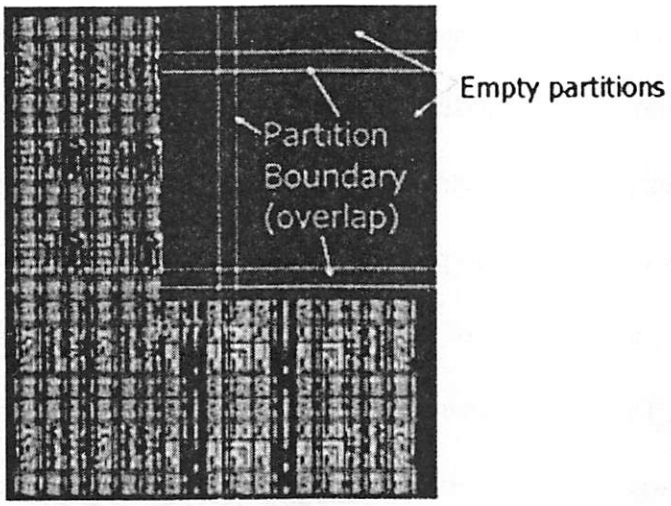


Figure 18: Layout divided into six overlapping partitions

7.4 *Geometry Processing*

7.4.1 Polygon Splitting

Complex input layout shapes are difficult to store, access, and process quickly inside of the matching loop, and for this reason they are split into simpler elements as a preprocessing step. Input shapes consist of rectangles, polygons, paths, and circles. Since each of these is either a type of polygon or can easily be converted into a polygon given a layout grid, explaining the procedure for the case of a polygon will be sufficient. The input polygons are split into geometric primitives, which consist of pixels, edges, rectangles, and triangles depending on the matching method used. The overall goal of the polygon splitting algorithm is to produce a minimal, non-overlapping set of smaller primitives that together cover the entire area inside the polygon and none of the area outside the polygon. It is relatively easy to convert polygons into bitmaps and edges since these operations are common in computer graphics and computational geometry, so only the more complex rectangle- and triangle-based splitting will be discussed here. Keil provides a good overview of polygon classifications and the methods of splitting polygons into smaller primitives [51], though special code was written for this research to split polygons for the application of pattern matching.

The choice of layout grid is critical here, since choosing too coarse of a grid can lead to discretization errors, single pixel gaps in the polygons after they are split, and misclassification of 45-degree angles as non-45 degree angles due to grid snapping of the edge endpoints. The grid size should be chosen to be at least as small as the shortest polygon edge in the input file in order to minimize the discretization error.

Though the algorithm will in theory work with self-intersecting and other special case polygons, these situations occur infrequently and the discussion of these cases will be omitted. The algorithms do not have any constraints on the layout geometry, but of course have not been tested on all possible geometries. One note on self-intersecting polygons is that processing is much easier if the intersection points are added into the list of polygon vertices if not already there. If polygons are preprocessed in this way, then no special cases are needed for the splitting of self-intersecting polygons.

7.4.1.1 Splitting a Polygon into Rectangles

Most of the polygons found in an integrated circuit mask layout are Manhattan, consisting of alternating horizontal and vertical line segments with no diagonal edges. Many layouts contain only Manhattan geometry. These are the easiest to process, and the splitting procedure has been optimized for this case. These polygons are first split into rectangles that are snapped to the layout grid.

The splitting algorithm proceeds by scanning through the points and locating the set of unique X and Y values that, if used as horizontal and vertical cutlines, will partition the polygon into a large number of smaller rectangles on a non-uniform grid. Each grid element that lies inside the polygon is extracted as a rectangle, and then merged with rectangles to the right and above to produce a small number of maximally expanded rectangles. In order to extract the rectangles, a binary edge matrix is built, where a value of '1' represents the presence of a vertical edge along that cutline segment. An in_poly binary flag is initialized to 0 and toggles each time a 1 is encountered in the edge matrix. Each X value of each row of the edge matrix is iterated through, and horizontal rectangles are extracted from the polygon for each consecutive Y value. The starting X value of the rectangle results from the location where the in_poly flag toggles from a 0 to a 1, and the ending X value results from a 1 to 0 toggle in the same row. Each rectangle is stored in an STL vector and possibly split into smaller primitives such as pixels and edges in a post-processing step.

If the bitmap algorithm of Section 6.2 is to be used, then the layer weight of each of the rectangles is added to the layer map through a scan conversion process such as that used in computer graphics [52]. The layer map is a bit-vector matrix, where each bit corresponds to the presence or absence of a layer. The layer map is converted into the 2D cell matrix prior to calculating the match values at each location. This leads to a cell matrix of floating point pixel values equal to the sum of the weights of every layer present at that pixel. Alternatively, if the edge intersection algorithm of Section 6.3 is used, then the rectangles are split into horizontal strips one pixel wide that lie between two opposite edges of each rectangle, and adjacent edge strips are merged into a single larger edge strip. The primitives are then sorted by Y position and then X position, and stored in memory or on disk.

7.4.1.2 Splitting a Polygon into Rectangles and Triangles

Polygons with diagonal lines pose a problem since they cannot be split directly into rectangles aligned with the X- and Y-axes. If the above algorithm were used, then some of the non-uniform grid elements would lie partially inside and partially outside of the polygon. There are two solutions to this problem: (1) Convert the diagonal edges into stair-step sequences of horizontal and vertical line segments discretized on the layout grid such as in scan conversion, resulting in a Manhattan polygon that can be split into a (potentially large) number of rectangles, or (2) Split the polygon into a small set of triangles as well as rectangles.

For practical purposes the triangles must be constrained to be right triangles; otherwise the processing of arbitrary triangles becomes too complex and computationally expensive, as explained in Section 6.5.1. Therefore, layouts with acute angles formed by two diagonal line segments inside of a single quadrant of the XY plane cannot be split into right triangles and muse be dealt with using method (1).

Polygon triangulation and tessellation are common to fields such as finite element analysis, mesh-based simulators, and computer graphics. There are a number of known polygon triangulation algorithms from the field of computational geometry [53]. Unfortunately, the odd properties of the pattern matching algorithms described here require right triangles combined with rectangles, and, unlike most triangulation algorithms, allow T-junctions. Most of the known triangulation algorithms do not satisfy the right triangle constraint, and the ones that do produce more triangles than necessary in efforts to remove small triangles or triangles with small angles. For this reason a custom triangulation algorithm was developed.

When method (2) can be used, the number of resulting primitives is usually much lower that when using method (1) due to the large number of rectangles required to accurately approximate a diagonal edge. Furthermore, if method (1) is used then the edges of these small approximating rectangles can lead to incorrect classification of the diagonal edge as a series of line-ends. The diagonal edges will also contain invalid inside and outside corners positioned at the corners of the approximating rectangles. Thus method (2) is generally preferred whenever possible.

The extraction of rectangles and triangles from a polygon is similar to the extraction of rectangles alone, with a number of additional algorithm steps discussed below. Figure 19 demonstrates the polygon splitting process.

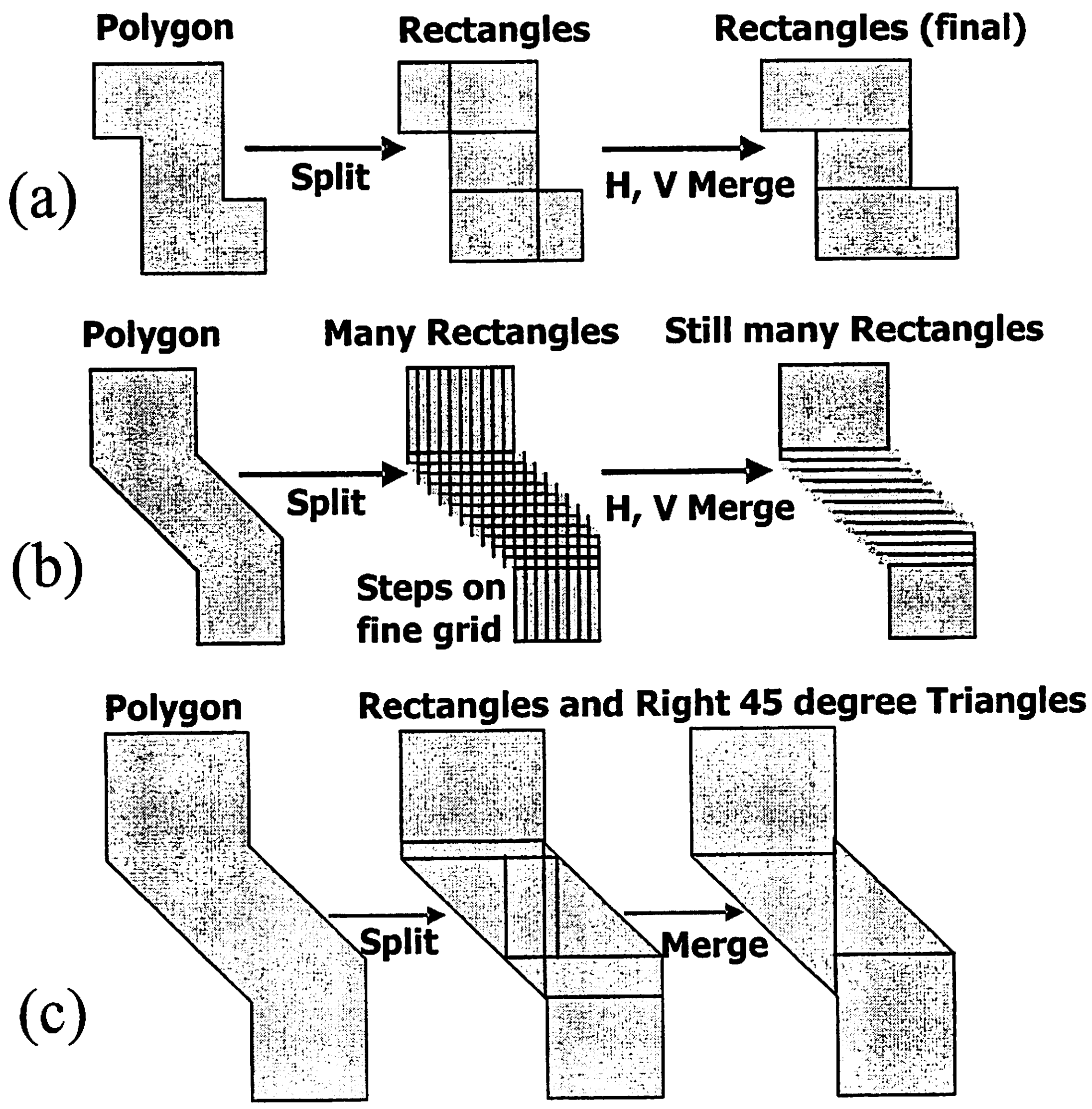


Figure 19: Splitting of polygons: (a) Manhattan polygon split into rectangles, (b) non-Manhattan polygon split into rectangles, and (c) non-Manhattan polygon split into both rectangles and triangles

Splitting a polygon into triangles is a common operation found in areas such as mesh triangulation for finite element analysis [54] and polygon-triangle tessellation in computer graphics [55]. However, the requirements for pattern matcher polygon triangulation are somewhat different, and the standard algorithms cannot be used. In this case the goal is to split a polygon into rectangles and triangles with a primary objective of minimizing the number of triangles and a secondary objective of minimizing the number of rectangles. All triangles must be right triangles, and, unlike in most triangulations, T-junctions formed by triangle edges are allowed. Furthermore, triangles resulting from self-intersecting polygons must not overlap. The general polygon splitting method employed here is to adaptively determine a minimal set of horizontal and vertical cut lines that exactly divide the polygon into rectangles and triangles. Choosing a minimal set of cut lines automatically leads to a minimal set of resulting rectangles and triangles.

Diagonal polygons are first split into a grid along all unique X and Y vertex values so that all horizontal and vertical polygon edges lie on a grid line and each grid rectangle contains only diagonal interior edges. Next, each grid rectangle is classified as containing one of the following types of geometry:

(1) Nothing,
(2) A rectangle that fills the entire grid region,
(3) A single triangle, or
(4) More complex polygon(s).

Case (1) requires no work and cases (2) and (3) result in a single rectangle or triangle being added to the output set. The polygon(s) in case (4) must contain at least one interior diagonal edge, and the grid region may contain more than one disjoint polygon. Case (4) is more difficult and requires a recursive call to the diagonal polygon splitting function. The sub-polygons are repeatedly split in this way until one of cases (1) through (3) is reached or all sub-polygons have zero area. Long, thin diagonal lines may be split into a large number of rectangles and triangles, so the result set is not bounded by the number of vertices in the polygon. However, these excessively high-aspect ratio polygons are rarely encountered.

Once the polygon has been split, an iterative merge stage is entered which compares combinations of adjacent rectangles and triangles for possible merging into a single larger shape. After a minimal set of rectangle and triangles is found, the set is added to the working shape set of the current partition. Once all of the polygons have been spit, the list of rectangles and triangles is iterated over to prune out any shapes that lie completely outside of the partition and to properly clip any shapes that partially overlap the partition.

7.4.2 Shape Merging

In order to reduce the number of primitives in the layout, some adjacent and compatible primitives on the same layer are merged into a single larger primitive. The following discussion deals with the merging of rectangles and triangles, since bitmap and edge merging is performed by simply adding the weights of overlapping pixels and edges into a single representative primitive. Triangles are only merged with rectangles within a polygon as discussed at the end of Section 7.4.1.2. In the global merge step, triangles are compared with opposing triangles in order to combine two triangles into a full rectangle.

The set of rectangles is first subdivided into smaller working regions, and an iterative quadratic-runtime algorithm is used to compare every shape in a region to ever other shape to see which can be merged. Two rectangles can be merged horizontally if they are adjacent (share opposite vertical sides) and have the same Y values. Two rectangles can be merged vertically if they share opposite horizontal sides and have the same X values. The merging proceeds until no rectangle can be merged with any other rectangle or an iteration limit is hit. Then the rectangles overlapping the boundary of the region are merged with rectangles overlapping the opposite boundary of the adjacent regions in a similar merge across the region boundaries.

7.4.3 Overlap Removal

In some cases, multiple shapes may overlap and an overlap removal step is necessary to ensure the contribution from overlapping shapes is not counted more than once. The overlap removal procedure can be thought of as a special case of OR-ing one layer with itself. Overlap removal is currently only supported for rectangles (16 cases) due to the large number of cases of triangle-triangle (256 cases) and triangle-rectangle (64 cases) overlap. The rectangle overlap removal procedure can be used either before or after merging, or can be inserted between two merge steps. The pattern matcher performs overlap removal before merging. It has the same basic computational complexity as rectangle merging because it also has to test all rectangles against all other rectangles in a quadratic algorithm on the same subdivision regions. For each pair of overlapping rectangles, a 16-way binary case split is used to determine how to modify one of the rectangles in order to remove the overlap. Depending on the case, one of the rectangle edges is moved in, one rectangle is split into multiple smaller rectangles, or a rectangle is removed.

7.5 Match Filtering

Quite often the user is only interested in matching on a particular layer or a particular type of geometry such as edges, line ends, and corners. It makes little sense to compute the match factor at points in the middle of a feature or far from a feature. Matching time is dependent on the number of points at which the match factor is computed and scales with resolution for line ends and edges, scales with resolution squared for unrestrained areas, and does not scale with resolution for corners represented by single points. The implementations and uses of geometric and layer-based match filtering are described below, in addition to coherency-based filtering for runtime improvement. Figure 20 demonstrates the dramatic reduction in required work due to pipelined filtering. These filtering algorithms are not easily applicable to the general FFT, statistical, and learning-based algorithms but work well with the rectangle algorithm.

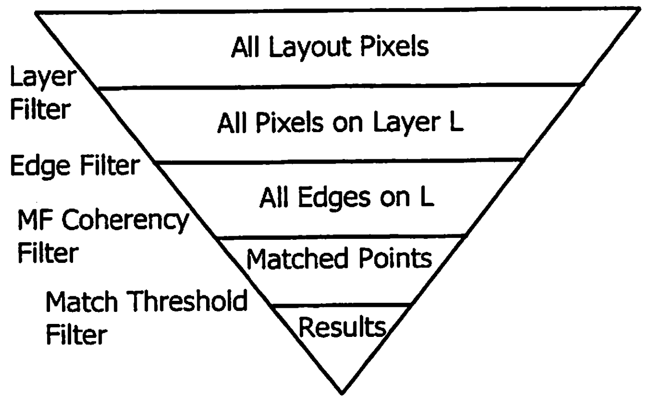


Figure 20: Multilevel filtering of match locations in the pattern matcher

At times the user may only wish to run the pattern matcher on part of a layout or a single cell in the layout or library. The pattern matcher supports the use of a bounding box to specify a coordinate range in which to match, and only geometry within that box is processed. Similarly, the user can force the pattern matcher to only match on a set or range of user-supplied pixel values regardless of what the geometry at that point is. The pattern matcher can also be instructed to only match on a single cell of a GDSII or CIF layout instead of the top cell(s).

7.5.1 Corners

The pattern matcher understands two types of feature corners: inside corners and outside corners. An inside corner is defined as an intersection of two polygon edges where the angular sweep from one edge to the other edge through the polygon covers more than 180 degrees. An outside corner is an intersection of two edges where the angular sweep is less than 180 degrees. Obviously, if the angular sweep is exactly 180 degrees then the point is not a true corner, and the geometry processing code removes these co-linear points from polygons before they are processed. A rectangle contains four outside corners, and an L-shaped polygon contains five outside corners and an inside corner. In fact, every properly formed Manhattan polygon contains four more outside corners than inside corners.

In some cases the user might want to only match on a certain type of corner, or an edge or line end between certain types of corners. The corner type of a rectangle or triangle can be inherited from its parent polygon before splitting, but this does not always yield the proper classification in the case where two or more polygons share the same vertices. In order to guarantee that all corners are correctly classified, each of the corners on every shape is preprocessed to determine its true status prior to entering the inner matching loop. For each corner coordinate, every rectangle and polygon that has a corner at that location is considered in computing the type of corner, if any. For example, if two rectangles meet at a point such that the top rectangle's lower right corner is at the same coordinate as the bottom rectangle's upper right coordinate, then the corner disappears and is flagged as neither inside nor outside in both rectangles.

7.5.2 Edges/Line Ends

Sometimes the user may wish to only match on edges in the layout, or to only match on line ends. For example, the user might want to match on the pixel at the center of all polysilicon line ends as defined by the midpoint of all lines less than or equal to a minimum feature in length that reside between two outside corners. It is possible to specify this exact set of match requirements in the pattern matcher. Edge classification must be performed once per rectangle or triangle and takes up to 20% of the total pattern matching runtime, but the classification can be reused for each additional pattern and thus edge processing time does not scale with the number of patterns. The pattern iteration loop is thus nested inside of the partition-processing loop so as not to discard the edge and corner information of the current partition's shapes.

Deciding whether the side of a rectangle or triangle is an edge or a line end requires knowledge of the length of the full edge between two opposing corners. The lengths of the original polygon edges are stored in the resulting rectangles and triangles as the polygon is split, but this length may not actually be the length of the true edge. The edges of two adjacent polygons may in fact meet to form a single larger edge, or an opposing polygon can meet partway along the edge and split a single edge into multiple segments. The common edge to two adjacent and opposite rectangles can even disappear completely if the rectangles are merged together. The actual length of rectangle or triangle's side, if it is even part of the true perimeter of the parent polygon, must be recalculated by processing all shapes with sides adjacent to the side of interest. The initial classification at the time of polygon splitting can be used to speed up this test, and after testing a shape the status of its sides and corners can be back annotated into the data structure so that the test does not need to be performed again later. The polygon shown in Figure 21 contains edges, line ends, inside corners, and outside corners.

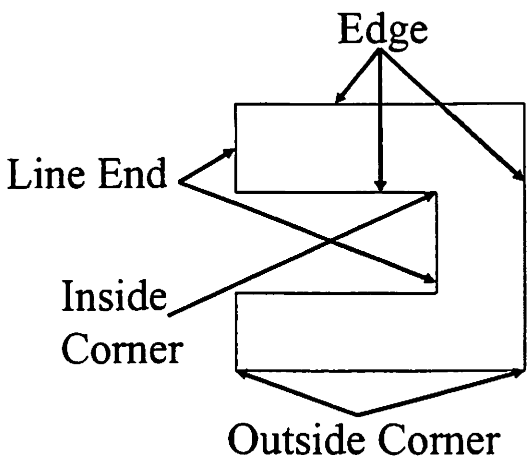


Figure 21: Polygon edge and corner classifications

7.5.3 Layers

Layer requirements are also possible in the pattern matcher. The user can optionally specify a single target layer, which may be different for each pattern. Normally, the edges and/or corners of each shape are iterated over to compute the match factor, depending on the type of edge, line end, and corner filtering. If the user requires a match only on a single layer such as METAL1, then only shapes on the target layer are iterated over. Other layers may still contribute to the match factor, but then cannot contribute to the locations at which the match factor is computed. Edge, corner, and other types of preprocessing steps are also only performed on the target layer(s) for efficiency purposes.

7.5.4 Match Factor Prediction Filter

The matching time of the rectangle algorithm was reduced by a factor of two to five by using an adaptive, spatial coherency-based match factor prediction and error bounds estimation to skip the computation of the match factor at points that are close to other locations of low value.

Let ΔMF_x be the maximum single horizontal pixel match factor change of pattern P that is possible with any layout. ΔMF is related to pattern smoothness and is small for slowly varying patterns. Thus, for any given layout where P is matched at location (X, Y) with match factor MF(X, Y), we have:

$$| MF(X,Y) - MF(X \pm 1, Y) | \leq \Delta MF_X$$

The minimum acceptable match factor threshold T is equal to the correlation factor times the best match found so far, or T = corr_factor*best_match. Any matches below this value will not be reported as pattern matching results. Now assume the match factor at (X, Y) has been calculated and is less than T, and match factor at (X+1, Y) is next to be computed. If the maximum possible match factor at (X+1, Y) is less than T, then point (X+1, Y) cannot be a result and thus the match factor computation can be skipped at this point. This condition is expressed as:

If ($MF(X,Y) + \Delta MF_X < T$) then skip (X+1, Y) and (X-1, Y)

The filtering procedure presented above can be generalized to any translation in a combination of X and Y directions by computing a matrix of match factor deltas $\Delta MF(i,j)$. Now the test can be generalized to:

If ($MF(X,Y) + \Delta MF(m,n) < T$) then skip all pixels at coordinates (X±m, Y±n)

This filtering step allows many of the expensive match factor computations to be avoided when testing the layout at dense intervals. However, the speedup achieved from this method is limited by the relatively large values of ΔMF. Now remember that ΔMF was determined for any layout and corresponds to the layout resulting in the highest match factor changes, which for typical patterns is nothing like a real integrated circuit layout. In reality the highest ΔMF actually found can be as small as one-fourth the calculated worst-case value. Unfortunately, it is impossible to determine the highest occurring ΔMF values without computing the match factor at every point in the layout, which would defeat the purpose of this speedup. The solution is to start with the conservative worst-case bounds for the ΔMF matrix entries, and dynamically adjust the expected highest ΔMF values as the layout is processed. As more low-valued ΔMF entries are actually calculated, the pattern matcher's confidence that these low ΔMF values are actually correct upper bounds is increased. The highest ΔMF values previously found in the layout provide a ceiling for the ΔMF entries used in filtering, and the values asymptotically approach this ceiling over time. Eventually, these values stabilize at the highest values of ΔMF actually occurring in this layout, which in theory provides the largest possible speedup with little chance of missed matches due to underestimating ΔMF values.

Thus the above speedup method has the ability to dynamically adapt to the pattern values and layout geometries. This filtering reduces the feature edge and line end points at which the match factor is calculated by a factor of around five, improving the overall runtime by a factor of two or more depending on the estimation parameters. The improvement is even greater when matching on a dense grid of points but is generally not helpful when matching on corners since the corners are too far away from each other to accurately predict the match factor using another corner match. The theory behind the match factor prediction filter is similar to the theory used in the bitmap algorithm compression of Section 6.2.1, in that both ideas involve the estimation of error incurred by approximating the match factor from other available information.

7.6 Layer Booleans

Some applications of pattern matching and certain methods of constraining the locations examined in a layout require Boolean and other layer operations (AND, OR, XOR, NOT, ANDNOT, EDGE, GROWBY) between two or more layers. These layer operations are needed to extend the pattern matcher to analyzing residual processing effects that involve operations on multiple layers, such as laser-assisted processing and reflective notching. The user is able to specify these layer operations in the layer definition file. The operations are performed on the spatially subdivided rectangle data of each partition. The results of the layer operations are used in the pattern matching inner loop and are possibly stored on disk or in memory for later graphical display of the generated layers. Layer Boolean operations are currently only supported for rectangles due to complexities of supporting these operations on triangles. Figure 22 shows the results of applying these layer operations to a simple two-layer layout. The various layer operations and their uses are explained below.

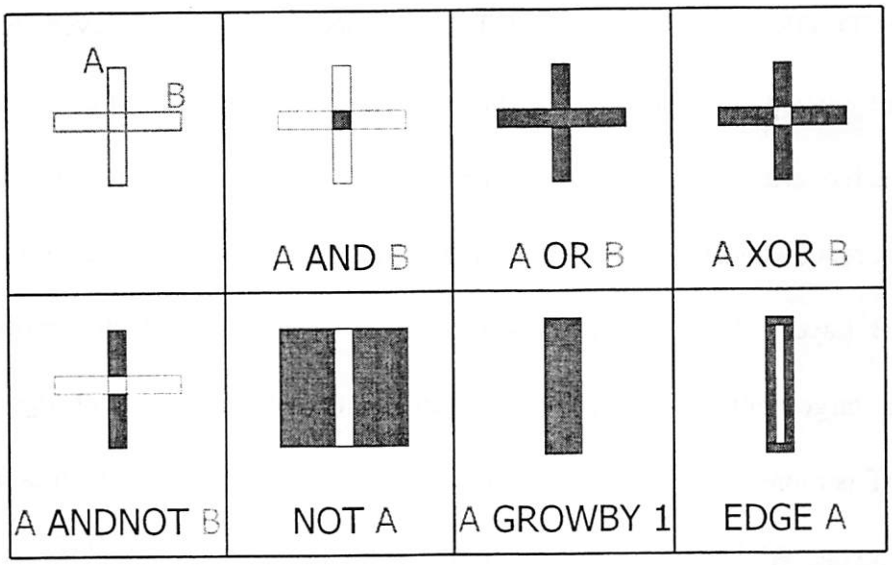


Figure 22: Illustration of layer Boolean and other operations

7.6.1 AND, OR, XOR, NOT, ANDNOT

A complete set of Boolean layer operations includes AND, OR, XOR, NOT, and ANDNOT operators. A subset of only two operators such as AND and NOT will suffice, but this rich set of layer Booleans allows for Boolean layer expressions involving fewer operators. The layer Boolean implementation was simple when using the bitmap algorithm since the Boolean operation could be applied at the bit level directly to the layer map matrix. Eight layers could be processed at a time by using a table lookup for each 8-bit byte in the layer map. The Boolean layer computation process for the rectangle algorithm is similar to the overlap removal and shape merging steps in that the collection of shapes is subdivided into small regions where a quadratic algorithm does not induce a large performance penalty. Efficient polygon Boolean operations are known [56], but it was decided that a more optimal way of performing Boolean operations was to work directly with the spatially subdivided rectangle dataset.

AND, OR, XOR, and ANDNOT are defined for two input layers A and B. The algorithms are implemented with custom computation kernels that test each rectangle in set A with each overlapping rectangle in set B. Layer AND produces a results set where each element rectangle corresponds to an overlap between a rectangle on layer A and a rectangle on layer B. Layer OR and the related N-way OR simply duplicate all shapes on layers A, B, and other targets into a new layer and then run the overlap removal algorithm. Layer ANDNOT is more complex and involves maintaining a results set which is initialized by duplicating the "AND" layer, and each of the "NOT" layers is subtracted from the results set by subdividing and removing rectangles. The layer XOR of A and B is basically implemented as (A ANDNOT B) OR (B ANDNOT A). The ANDNOT algorithm is integrated into the XOR function for improved efficiency over using a function call, and again the OR operation is implemented with an overlap removal step. Finally, the layer NOT operation on layer A is computed as 1 ANDNOT A by initializing the "AND" layer of the ANDNOT Boolean to be a rectangle covering the entire region of the layout on which the NOT operation is computed.

Layer OR simply involves overlap removal, layer AND requires computing the overlap between layers, and layer XOR and layer NOT can be built from layer ANDNOT. The layer ANDNOT algorithm is the only missing component and will now be described. Let $A = \{a_0,...,a_M\}$ denote the set of M shapes and area covered by the "AND" layer and $B = \{b_0,...,b_N\}$ denote the set of N shapes and area covered by the "NOT" layer. The resulting output set R consists of rectangles covering all of A that is not included in B, or $R = A - B$. Let Q be a queue of (rectangle, index) pairs which initially contains all M rectangles in A with indices set to zero, $Q_{init} = \{\{a_0, 0\},...,\{a_M, 0\}\}$. If ANDNOT is being used for layer NOT, then Q will initially contain a rectangle equal in size to the entire region.

The algorithm proceeds by removing (rectangle, index) pairs $\{r, I\}$ from Q until Q is empty. Each $\{r, I\}$ is tested for overlap with each of the elements b_i of $b_I,...,b_N$. If overlap is found, then several cases are possible:

(a) r is completely inside of b_i,
(b) $s = r - b_i$ is a single rectangle, or
(c) $S = \{s_0,...,s_P\} = r - b_i$ results in P rectangles, where $P > 1$.

If case (a) occurs, then r can be eliminated and the algorithm goes on to remove another pair from Q. If case (b) occurs, then r is replaced by s and the next rectangle in B, b_{i+1}, is tested for overlap. If case (c) occurs, then r is replaced by s_0 as in case (b), and the remaining P-1 rectangle(s) $\{s_1,...,s_P\}$ are paired with the next index of b and are inserted into Q as $\{s_j, i+1\}$

pairs. After rectangle r is checked against the last element of B, b_N, it is inserted into R. When Q is empty, then result set R is output and the algorithm finishes. This algorithm is guaranteed to terminate with the correct set of rectangles, and the maximum size Q will reach is linear in max(M, N).

7.6.2 Grow-by

Layer grow-by increases or decreases the size of a polygon by moving the edges closer or further from the center of the polygon and then possibly performing overlap removal. This operation is typically used to expand polygon boundaries for EDA applications such as Design Rule Checking (DRC). For example, a 10 by 10 pixel rectangle grown by 2 pixels on each edge will result in a 14 by 14 pixel rectangle at the same center point. Layer grow-by is useful for pattern matching applications such as reflective notching, as discussed in Section 9.3, where a fixed-width border around the edge of a layer must be extracted by growing the edge. Negative grow-by can also be used to filter out small polygons. A negative grow-by followed by a positive grow-by of the same amount will remove all polygon features less than twice the grow amount in size.

Positive grow-by operations may not work properly if the grow amount causes rectangles or triangles to extend beyond the boundaries of the layout or partitions. In addition, large negative grow amounts may cause a single large polygon to be separated into a number of smaller polygons, and in some cases this results in errors in the rectangle and triangle representations of the split polygons.

Negative grow-by is much more complex then it first appears because several interesting things can happen. If the grow amount is large enough, polygons can both disappear and become broken up into several disjoint polygons. In addition, bridging rectangles must be created in certain situations to connect two rectangles that were adjacent prior to the negative grow-by operation but moved apart as a result of the grow-by. The addition of these bridge rectangles becomes difficult when several rectangles come together at the same point, or when rectangles less than two times the grow amount in one or both dimensions that would normally disappear are clustered together such that the combined shape is large enough to be present after the grow-by. The negative grow-by algorithm involves recursive calls that add these small bridge rectangles on the resultant layer, though certain odd configurations of rectangles, especially those with long adjacency chains and circular dependencies, can result in an infinite recursive loop. A recursion limit has been added to break the loops, but this results in small holes in some of the output shapes. A better solution probably involves storing the adjacency lists of rectangles or perhaps operating on the input polygon set instead of the post-split rectangles, but this involves significant changes to the code and has not been implemented.

7.6.3 Edge Extract

The edge extract operation is used to generate a new layer two pixels in width along the edges of an existing layer. This edge layer can then be expanded by a grow-by to form a strip surrounding the edges of the source layer, for use in applications that require matching on polygon boundaries. This operation is more difficult than it may seem due to the existence of internal edges in the rectangle dataset that come from shared edges of adjacent rectangles. An edge classification algorithm similar to the one explained in Section 7.5.2 is used to determine which sections of a rectangle's sides are true edges and to add new rectangles representing those edge segments only. The overlaps that occur at the corners and other locations between these edge rectangles must then be removed.

7.6.4 Layer Operation Complexity

Table 4 compares the per-region computational complexity and resulting set sizes of the various layer operations presented in this section. Capital letters denote the layers and lowercase letters denote the corresponding set sizes. The variable N represents the iteration count of iterative algorithms, which is usually limited to a maximum of around ten. C is a small operation-dependent constant factor of around two to four. Set sizes and computational complexities are worst-case bounds. Each of the input layers is assumed to be merged into a minimal set and overlap free. Layer OR, positive grow-by, and edge extract require final quadratic runtime overlap removal steps denoted by the "=>", which usually dominates the runtime. The computational complexity is highly dependent on the actual size of the results set for most of the algorithms, but is in general quadratic in the input set size. The worst-case size of the results set is typically much higher than the average case, typically quadratic as compared to linear in the input size. The average results set size is roughly the same size as the input set in most cases where the shapes do not significantly overlap between layers. The worst-case is often achieved when layer A consists of vertical lines and layer B consists of horizontal lines such that all shapes on layer A intersect with all shapes in layer B, which leads to the a*b term common to many of the resulting set sizes.

Operation	Inputs	Resulting Set Size	Computational Complexity
Merge	A	a	$N*a^2$
Overlap Removal	A	$(a+1)^2/4$	a^2
AND	A, B	a*b	a*b
OR	A, B	(a + b) => a*b	$(a + b)^2$
N-way OR	A, B, … N	Σ(a+…+n) => $(\Sigma(a+\ldots+n)+1)^2/4$	$(\Sigma(a+\ldots+n))^2$
XOR	A, B	C*a*b	$\max(a, b)^2$
ANDNOT	A , B	C*a*b	$\max(a, b)^2$
NOT	A	C*a	a^2
Pos. Grow-by	A	a => C*a	a^2
Neg. Grow-by	A	C*a	a^2 or more
Edge Extract	A	4*a => 4*a	a^2

Table 4: Complexity and resulting set size of layer operations

7.7 Match Proximity Factor

The match factor is usually slowly varying across the layout, and when calculated at a dense set of points on a fine grid we would expect to see groups of high match factors clustered together. The user may in fact want to more evenly spread the match factor results across the layout so as to reduce the number of matches found while still locating all generally problematic areas. It may be desirable to remove all match factors within a certain pixel proximity to higher matches, since whatever is done to handle a high match factor will likely also fix the high match factors occurring several pixels away. The final results are then guaranteed to be separated from each other by at least the proximity factor. The pattern matcher implements this feature through a user-define proximity factor that filters out all lower matches within a given distance of a higher match.

In order to account for the loss of match results, the pattern matcher internally increases the number of stored matches by a factor of ten or more. However, the match proximity factor still sometimes reduces the number of matches found below the user requested number. The number of matches filtered due to the proximity factor cannot be determined in advance, and storing too many matches could slow down the match queue operations. For this reason, it is often useful to show more results than are actually necessary and to have the user truncate the match list after a certain number of results or match factor threshold is reached.

7.8 Parallel Pattern Matching Algorithms

The pattern matcher was initially developed as a sequential program, but it is easily modified to take advantage of parallel computational resources for reducing runtime on large inputs. A typical distributed parallel cluster is comprised of a number of computation nodes that are networked together, each of which contains a number of processing elements and shared memory. The input layout can be spatially subdivided among nodes and processors, where the nodes communicate through the Message Passing Interface (MPI) [57][15], and the processors communicate through POSIX threads (Pthreads) [58][16]. MPI is a method of distributed communication that can be used to spread the pattern-matching job over a large number of machines linked by a fast network. MPI is not available on all machines and is enabled by a compile-time definition in the pattern matcher Makefile. Multiple Pthreads are run on shared memory machines with more than one processor. In fact, any thread library that supports mutexes or other locks can be used for parallelization.

15. http://www-unix.mcs.anl.gov/mpi/

The parallel pattern matcher was tested on two parallel computing clusters. The CITRIS cluster, a collection of dual processor 1.3GHz Itanium2s connected by gigabit Ethernet, is installed at the Electrical Engineering and Computer Sciences Department at the University of California, Berkeley. Seaborg, a large cluster of 16-way 300MHz IBM Power3 processors, is one of the National Energy Research Scientific Computing Center (NERSC)[17] machines.

The actual steps of the sequential pattern-matching algorithm are shown in Figure 23. The left side of the figure lists the pattern matching steps and the color code used in this figure and in Figure 24. The right side of Figure 23 shows the breakdown of runtime, and is approximately to scale for typical inputs. The actual matching step takes about half the runtime, and is fortunately easy to parallelize. The database operations take about 25% of the runtime and can be parallelized with more difficulty. The input file read takes a significant part of the runtime for relatively flat layouts, but could not be completely parallelized. Finally, the other steps are very fast and are probably not worth parallelizing.

16. http://www.humanfactor.com/pthreads/

17. http://www.nersc.gov

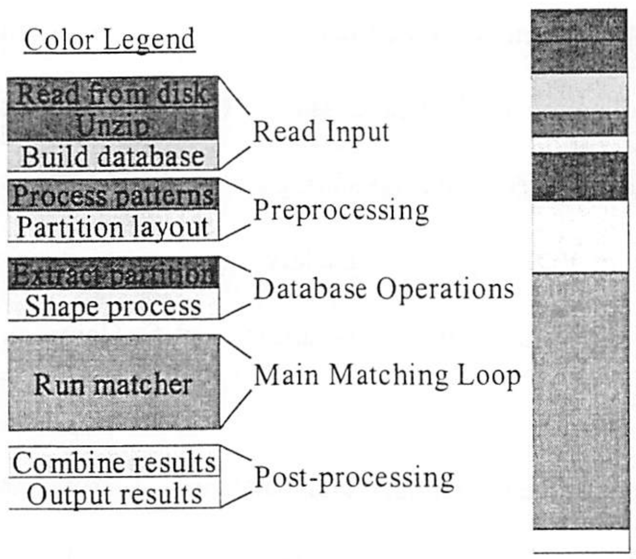


Figure 23: Pattern matching algorithm steps and breakdown of sequential code execution

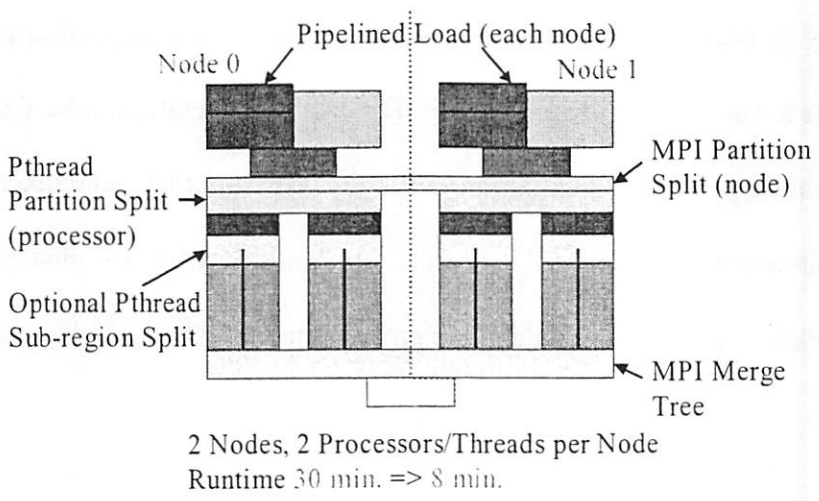

Pipelined Load (each node)
Node 0
Node 1
Pthread Partition Split (processor)
MPI Partition Split (node)
Optional Pthread Sub-region Split
MPI Merge Tree
2 Nodes, 2 Processors/Threads per Node
Runtime 30 min. => 8 min.

Figure 24: Breakdown of parallel pattern matcher algorithm

Figure 24 explains how the parallelization was accomplished. The colors correspond to the color code in Figure 23, and the heights of the blocks correspond to approximate runtimes of those steps. The parallel algorithm involves spatially subdividing the layout in both the X and Y directions and processing the partitions in parallel. The layout is already partitioned to conserve memory, but parallelization of the partitions is not easy. The partitioning is somewhat complex since the partitions must overlap and load balancing may be difficult due to the non-uniformity of the polygon distribution. There is no clear way to load the compressed hierarchical layout database in parallel or efficiently distribute the storage due to complex dependencies and overlaps between the cells. Thus each node must load its own copy of the hierarchical database.

The final algorithm uses an approximation method that is based on previously computed statistics, and it was initially thought that these values would have to be communicated between the processes at various intervals in order to effectively use the approximation algorithm. However, the approximation algorithm reaches a steady state after a few seconds, so each processor will independently reach the same value very quickly. The problem became much easier once it was determined that this communication was unnecessary.

Since the computation was expected to dominate over the communication, it was expected to work well on a distributed memory system using MPI. Most of the code blocks were untouched since the new parallel code went into the spatial subdivision algorithms and the outer loops. However, a significant amount of code change was required in order to make the pattern matcher thread-safe.

The parallelization was accomplished in two steps. First, each node loads the entire database from disk and spatially partitions it into several hundred areas. Node N processes every partition P where (P % Numnodes) == N. Then, the partition is further subdivided into regions, where each processor in node N works on its own set of regions. MPI was used for the inter-node communication, and Pthreads was used for communication within each node. This partitioning strategy is illustrated graphically in Figure 25 for the combined MPI and Pthreads case, with four nodes each containing four processors. The actual time taken to process a partition is highly variable due to the non-uniform distribution of polygons in an integrated circuit layout. However, if each node and each processor are assigned a large number of partitions that are evenly distributed over the entire chip area, then the overall variance per processor will be small. This leads to good load balancing across symmetric processing elements when the number of partitions is much larger than the number of processors.

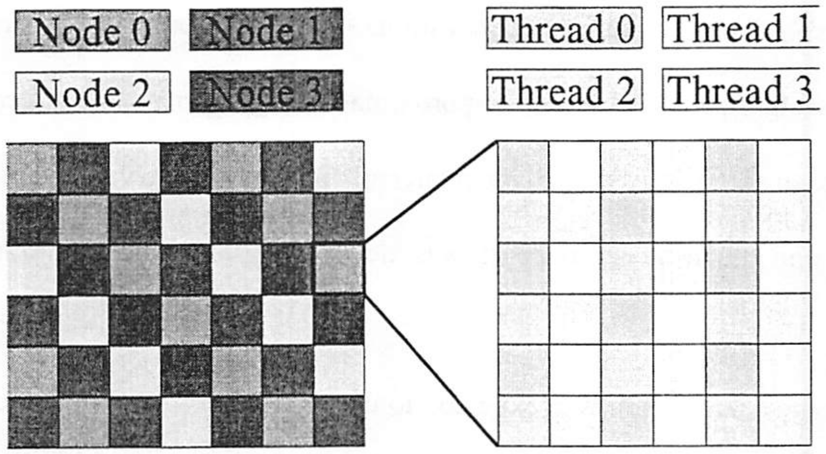


Figure 25: Spatial layout partitioning strategy for MPI and Pthreads with four nodes and four processors/threads per node

This use of Pthreads will be referred to as the "inner" loop version. The problem with the inner loop version is that the procedure of further subdividing the partition into sub-regions involves processing that takes as much as 10% of the total runtime. An alternative method, named the "outer" loop version, involves distributing the partitions among processors using a round-robin algorithm similar to the node partitioning method. The disadvantage of the outer loop method is that it requires several partitions to be in memory at once, one for each processor. However, this was not a problem in the test examples since both clusters had adequate memory for these small cases. The largest test run required only 577MB of RAM on a 64-bit machine.

Initially, all processors loaded the entire layout database, ran on their partitions, and sent their results to processor 0. Sending all of the results to processor zero is inefficient when the user asks for a large number of results, so this method was replaced with a binary tree-based results merge. A simple four-processor case of this binary tree is shown in Figure 26. In the first phase, node 1 sends its results data to node 0 for merging, and at the same time node 3 sends its data to node 2. In the next merge level, node 2 sends its data to node 0, and then node 0 outputs the final results. This binary tree merge algorithm works well even for 32 processors (5 levels), with a merge time of less than a second.

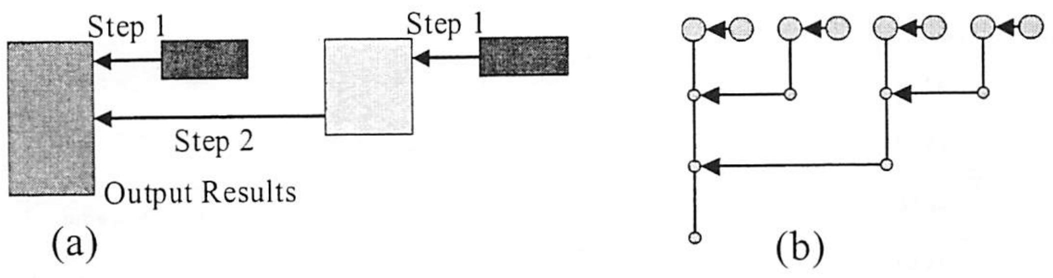


Figure 26: Binary tree results merge for (a) four and (b) eight processors

7.8.1 MPI and Parallel Partitioning

MPI results for the CITRIS cluster are shown in Figure 27. The parallel pattern matcher has a high parallel efficiency, achieving over 28X speedup on 32 nodes for the 25 min. active_test run, excluding the MPI startup time. The total runtime was reduced from 25 minutes to only 51 seconds. The match time scaling is near the theoretical limit, though there is a large MPI startup time of several minutes when using many nodes. This MPI startup time will not be as much of an issue when processing larger files that take hours to run sequentially and tens of minutes in parallel. This speedup was actually achieved with only one processor per node, which does not utilize all of the processing resources. However, the matching can also be partitioned using multiple Pthreads, and a 16 node run using both processors (two threads) achieves a speedup of over 29. It appears as though pure MPI and MPI + Pthreads work equally well, but MPI + Pthreads is preferable because it uses all available processors and less memory (only one database per node). The reason the speedup is not perfect is due to several issues, most notably the fact that the layout load and preprocessing is not done in parallel and has a small constant time penalty regardless of the number of processors.

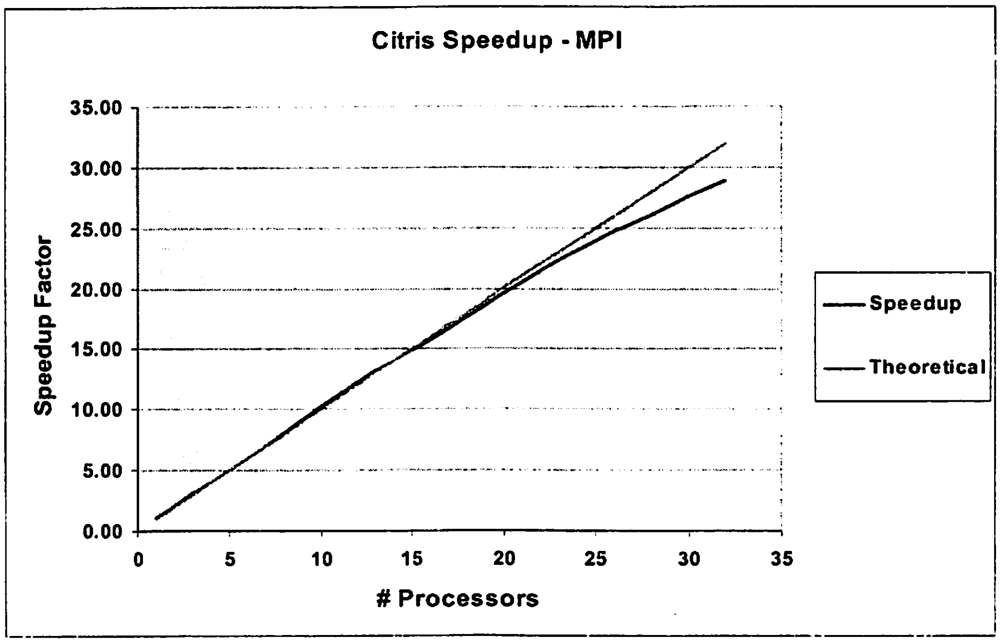


Figure 27: MPI runtime scalability results for the CITRIS cluster showing a near perfect speedup of 28X on 32 nodes

Figure 28 shows the MPI speedup on Seaborg for two different configurations. These numbers are for a single 16-processor node and multiple MPI tasks per node. It was found that MPI worked better than Pthreads on Seaborg nodes, but this is due to Pthread memory allocation problems as discussed in the next section. Seaborg had much longer runtimes than CITRIS on the standard benchmark input file, and could not be run in debug mode. A smaller example was used on Seaborg to achieve more reasonable runtimes. An example that used less than 128MB of memory was chosen so that interactive jobs could be run to avoid the difficulties of the queuing system. The first configuration involved starting an MPI job that ran on multiple nodes, using only one processor per node. The scaling was still quite good, giving a speedup of over 11 for 16 nodes, but it is not as good as on CITRIS due to the increased ratio of preprocessing time to match time required for this smaller example. In this case, significant time was spent in the pattern pre-integration, which has not been parallelized. The second configuration involved running MPI on the processors within a single node. This configuration did not scale as well due to memory contention within the node and the time taken for a single node to load multiple copies of the file from disk.

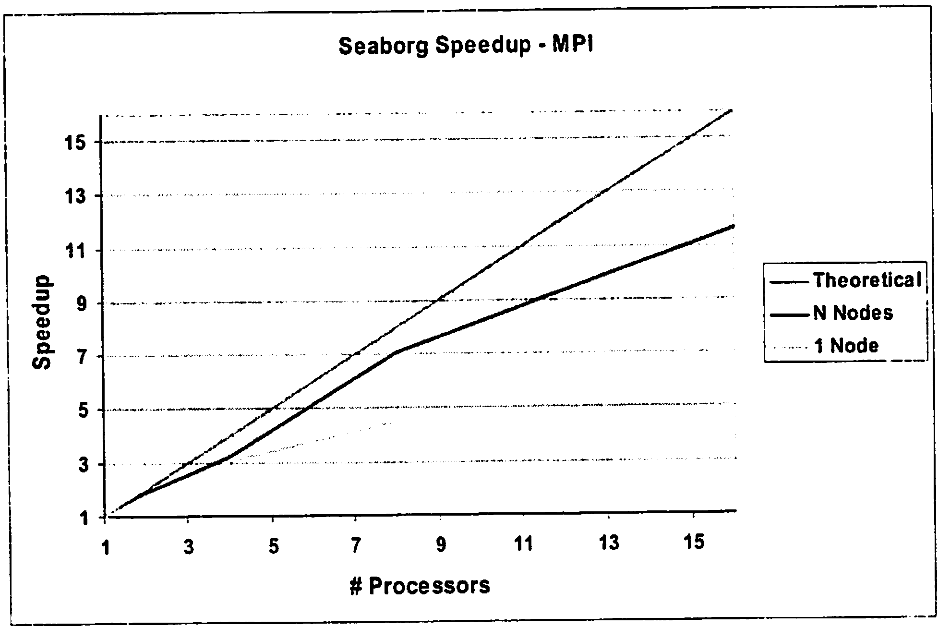


Figure 28: MPI runtime scalability on the Seaborg machine at NERSC

The parallel pattern matching scales extremely well on eight or more processors even for runs that require only a few seconds of runtime. There are still areas that need work, such as parallel pattern pre-integration and parallel results processing, but the majority of the processing time has been parallelized. The high computation cost and low communication cost of the MPI parallelization strategy leads to near perfect speedup that is only limited by load balancing issues, file load and preprocessing time, and MPI startup overhead.

7.8.2 Pthreads Dual Processor Comparison

The core pattern matcher executable has the capability of splitting the pattern matching work among POSIX threads (Pthreads). In fact, any thread library that supports mutexes or other locks can be used for parallelization. Multiple threads are run on shared memory machines with more than one processor.

The performance of Pthreads was characterized on various dual processor systems in order to determine what the scaling limitations were. Unfortunately, it was not possible to achieve good performance with Pthreads on Seaborg without significantly changing the code, as the runtime actually increased with the number of threads. It is possible that the threads were fighting with the memory system when allocating small amounts of memory inside of the Standard Template Library (STL) vector resize (doubling array), which occurs when constructing partial vertex arrays inside of the recursive polygon splitting code. It is not clear how to remedy this problem, since the optimal vector allocators are not available with the version of the STL installed on Seaborg.

Figure 29 shows the relative performance advantages that come from utilizing both processors of a dual processor machine. As explained in the previous section, MPI can be run on both processors to achieve a near perfect speedup of 1.95 on a CITRIS Itanium2 node. This is because each processor builds the entire database and there is no memory sharing. If Pthreads are used instead of MPI, then the speedup is slightly lower due to memory contention and time spent waiting at mutex locks. However, the memory requirements are lower since only one copy of the database is stored, though two partitions are also stored. If the Pthread split is done later in the code (inner loop) at finer sub-region granularity, the memory is reduced further but the speedup falls since a smaller portion of the processing is done in parallel. The database access is still sequential.

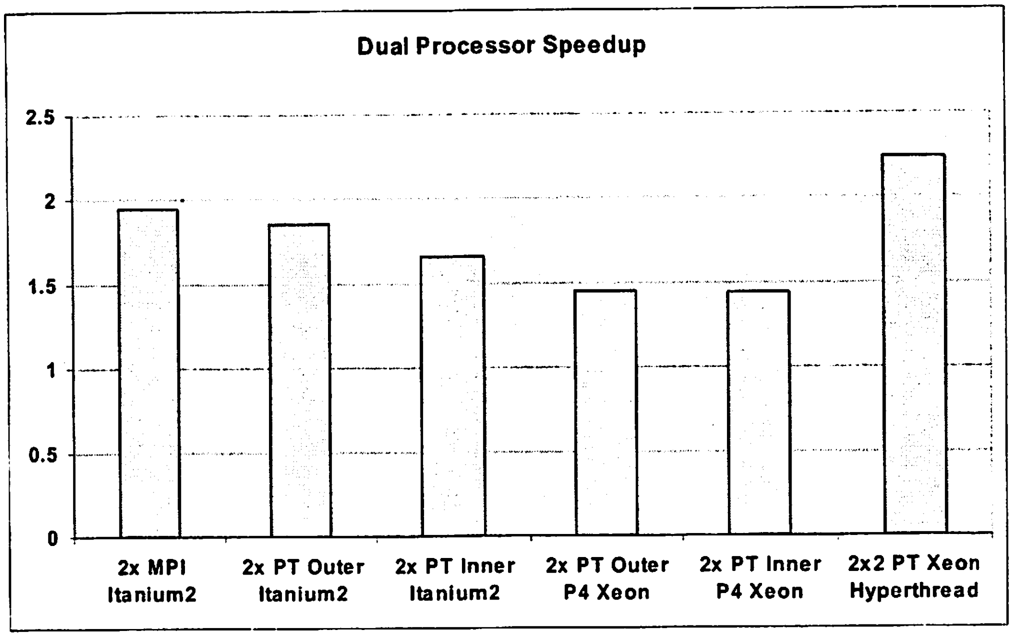


Figure 29: Pattern matching performance on several dual-processor architectures

A 3.2GHz Pentium 4 Xeon with hyperthreading was tested as well as the CITRIS and Seaborg nodes. Both Pthread algorithms resulted in a speedup of less than 1.5 due to memory contention. Memory bandwidth is more of a bottleneck on the Xeon system than on an Itanium2 since the processing speed to memory bandwidth ratio of the Xeon is so much higher. Starting four computation threads on the Xeon instead of two results in an additional speedup, bringing the total speedup to 2.25. This extra speedup comes from the hyperthreading technology in the Xeon processor, which allows a single processor to run two thread instruction streams at once. Hyperthreading is not as good as having two real processors, but it does help somewhat in this case. The Xeon is very fast at sorting the integer data in the database operations, and doesn't suffer from as much slowdown when using the inner Pthreads split instead of the outer split.

7.8.3 Dynamic Load Balancing Queue

Partitioning the layout into many small regions and dividing them among processors works well in most cases. Each processor knows which partitions it processes in advance, and no communication is necessary until the merging stage at the end. However, if the number of partitions is small, the layout geometry distribution is extremely non-uniform, the processors are unequal, or some of the processors are loaded with other jobs, then load balancing becomes an issue. Static load balancing in this case leads to the fast processors finishing their partitions early and having to wait for the slow processors to finish before performing the results merge. Dynamic load balancing is easy when using Pthreads; each thread simply takes the next available partition from a queue and increments the shared memory partition counter inside of a lock. This method ensures that all partitions are processed and none are matched on more than once. That way each processor accepts more work only when it has finished its current job, allowing faster processors to work their way through a larger number of partitions.

Dynamic load balancing in MPI is much more difficult due to a lack of shared memory for the partition counter. Node 0 runs both a client thread and a server thread, while all other nodes run only a client thread, as shown in Figure 30. The server thread spends the majority of its time in an idle state waiting for partition requests, so it can be run on the same processor as a client thread without significantly slowing the client thread. Node 0's client communicates with node 0's server through shared memory so that node 0 can run the pattern matching process and simultaneously wait for other nodes to request new partitions to process. This shared memory communication is necessary since two threads running on the same processor may deadlock if they both try to send or receive MPI messages at the same time. Each client node other than node 0 sends an MPI request to node 0's server as soon as it is finished processing the current partition. Node 0's server returns a unique partition ID from the queue for that client to process, or a magic number to signify the end of the partition dataset. That way, each node is constantly processing a partition, and slower nodes will simply submit fewer requests to the server and thus process fewer partitions. If a node contains more than one processor, then either each processor can individually communicate with the server over MPI or the processors can communicate with a separate MPI communication thread running on the client node. The former approach has been chosen for the final parallel pattern matcher implementation. The communication system allows multiple processor threads on the same client node to simultaneously ask the server for partitions, and each thread will eventually receive a partition. The partitions may be processed out of order, though this is acceptable due to the independence of the partitions.

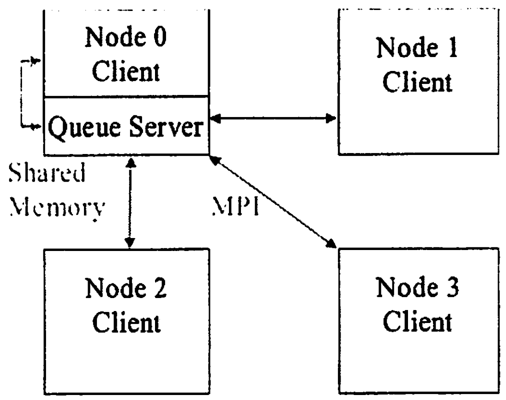


Figure 30: Dynamic load-balancing queue client/server model including both MPI and Pthreads communication

The effect of the dynamic load balancing queue was tested by observing the elapsed time between when the first processor reached the synchronization point at the results merge tree and the time the last processor reached that point. Perfect load balancing means that all processors reach the synchronization point at the same time. Figure 31 shows the runtime results of pattern matching averaged over several runs on the same input. The processors used are a combination of slow (900MHz) and fast (1.3GHz) Itanium2s. The horizontal axis represents the number of slow nodes among the eight running. The runtime increased as the ratio of slow nodes to fast nodes increased, as expected. The synchronization wait time increased sharply from almost zero to 43% of the total runtime with the addition of a single slow node, since now all of the fast nodes have to wait for the slow node to finish. There is no communication time in Figure 31 because the static load-balancing algorithm does not require communication. It is interesting to note that there is a load balancing issue even when using all fast nodes. This is a result of resource contention as some of the fast nodes were likely running processes for other users. The imbalance due to resource contention results in increased wait time at the merge point, just as if the processors are running at different speeds. This further justifies the need for dynamic load balancing, since in many real world situations the user will not have complete control of the machine resources.

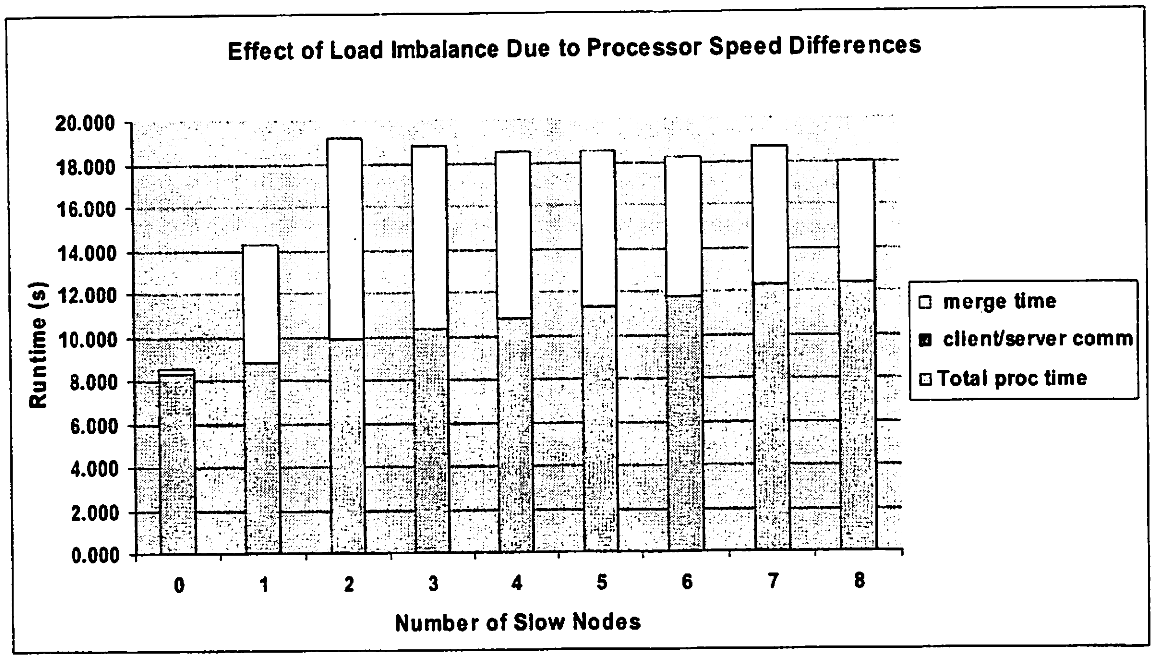


Figure 31: Parallel pattern matcher runtimes with static load balancing only

The results of running the dynamic load-balancing scheme as described above are shown in Figure 32. The wait time at the synchronization point has dropped to nearly zero, showing that the dynamic load-balancing algorithm is doing its job. The communication cost, shown in magenta, is less than 1% of the total runtime for all test cases. Finally, the wait time before synchronization can never be completely eliminated due to the granularity of the partition size and the fact that different partitions require different amounts of work. The processing time per partition is usually around one second, which is an upper bound on the theoretical best dynamic load-balancing algorithm. This dynamic load-balancing queue virtually eliminates the load-balancing problem and allows the pattern matcher to be run on heterogeneous machines and to adapt to heavily loaded systems.

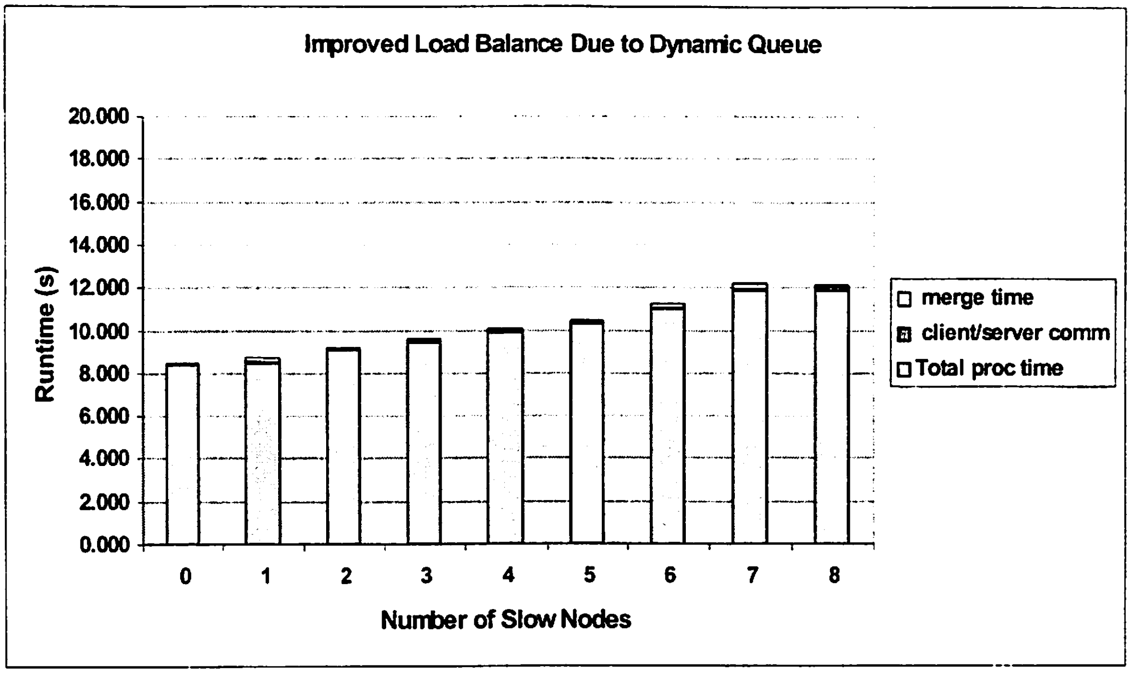


Figure 32: Parallel pattern matcher improved runtimes when using dynamic load balancing

8 Pattern Matching Experimental and Performance Results

Cesar Garza has pointed out the importance of aberration effects on printed features and the need to take aberrations into account during the design and manufacturing process [31]. This chapter examines the physical effects of aberrations on the printed image and characterizes the magnitude of aberration effects relative to optical proximity effects as corrected by OPC. The lens aberration idea proved to be a useful application of pattern matching that required the development of a pattern generation system, efficient full chip matching algorithms, and a procedure for linking results to other, more rigorous simulators. The automatic extraction of geometry for SPLAT aerial image simulation provides an ideal means to perform more rigorous analysis on the geometries found through pattern matching.

This chapter illustrates a number of example matching runs and compares the results to expected and simulated aberration effects, verifying the accuracy of the pattern matching approach. In addition, runtime and resource requirements are provided for a range of inputs from small layout clips to full chip mask layers containing over a hundred million polygons. Performance results show that the pattern matcher does meet its goal of processing a full chip in under an hour on a standard computer.

8.1 Experimental Conditions

Unless otherwise stated, the amount of aberration chosen for the simulations herein was 0.025 waves RMS, corresponding to a good lens with a Strehl ratio of 0.975. The partial coherence factor σ was chosen to be 0.3. The four aberrations chosen for these experiments were balanced coma (cos), coma (sin), high order coma (cos), and spherical. This list includes both even and odd aberrations, a similar pair, and a pair differing by a 90-degree rotation. A pattern matcher resolution of 0.06 λ/NA was calculated to match the generated pattern dimensions and to minimize the alignment error. Match factors were calculated using the pattern matching software system described above. Intensity changes where measured from the SPLAT simulation results of a cutline running perpendicular to and centered on the edge or line end of interest. Line-edge and line-end shifts were derived from the changes in intensity and $\Delta I/\Delta X$ slopes at the edges of the mask regions using a 0.3 intensity threshold.

It is important to note the difference between positive and negative match factors. Positive match factors result in increased electric field intensity and therefore line edge shifts in one direction, say to the right. Negative match factors result in decreased intensity, and therefore line edge shifts to the left. The actual match results should consider the absolute value of the match factor when ranking locations based on sensitivity to aberrations or other processing effects.

Most of the JPEG screenshot images in this chapter were taken with the integrated pattern matcher graphical display and with Display3D.

8.2 Pattern Matcher SPLAT Verification Procedure

Figure 33 presents a graphical flow of the interface between the pattern matching system and the SPLAT simulator. First, the pattern matcher is run on the coma (cos) pattern to determine the layout location that is most sensitive to coma. Notice that the underlying

PSM layout in Figure 33a correlates well to the actual pattern shape. The output of the matcher is then automatically extracted to SPLAT format. A Drawmask plot of the SPLAT file generated from the match in Figure 33a is shown in Figure 33b. The 0.3 intensity cutlines of the SPLAT simulation results for that same input geometry with and without aberrations are shown in Figure 33c. It can be seen from this plot that the addition of 0.1 waves of RMS coma (cos) widens the printed line and shifts it to the right. Figure 33e compares contour plots of the aberrated and unaberrated cases. Notice how coma makes the printed line in the center of the contour plot narrower (contour lines closer together) and shifted to the left. Figure 33d presents a 3-dimensional wire frame plot of the same contour generated by Display3D, where the low areas of the mesh would normally print as lines. The effects of coma may cause the line to short with another line to the right of it, leading to a faulty circuit.

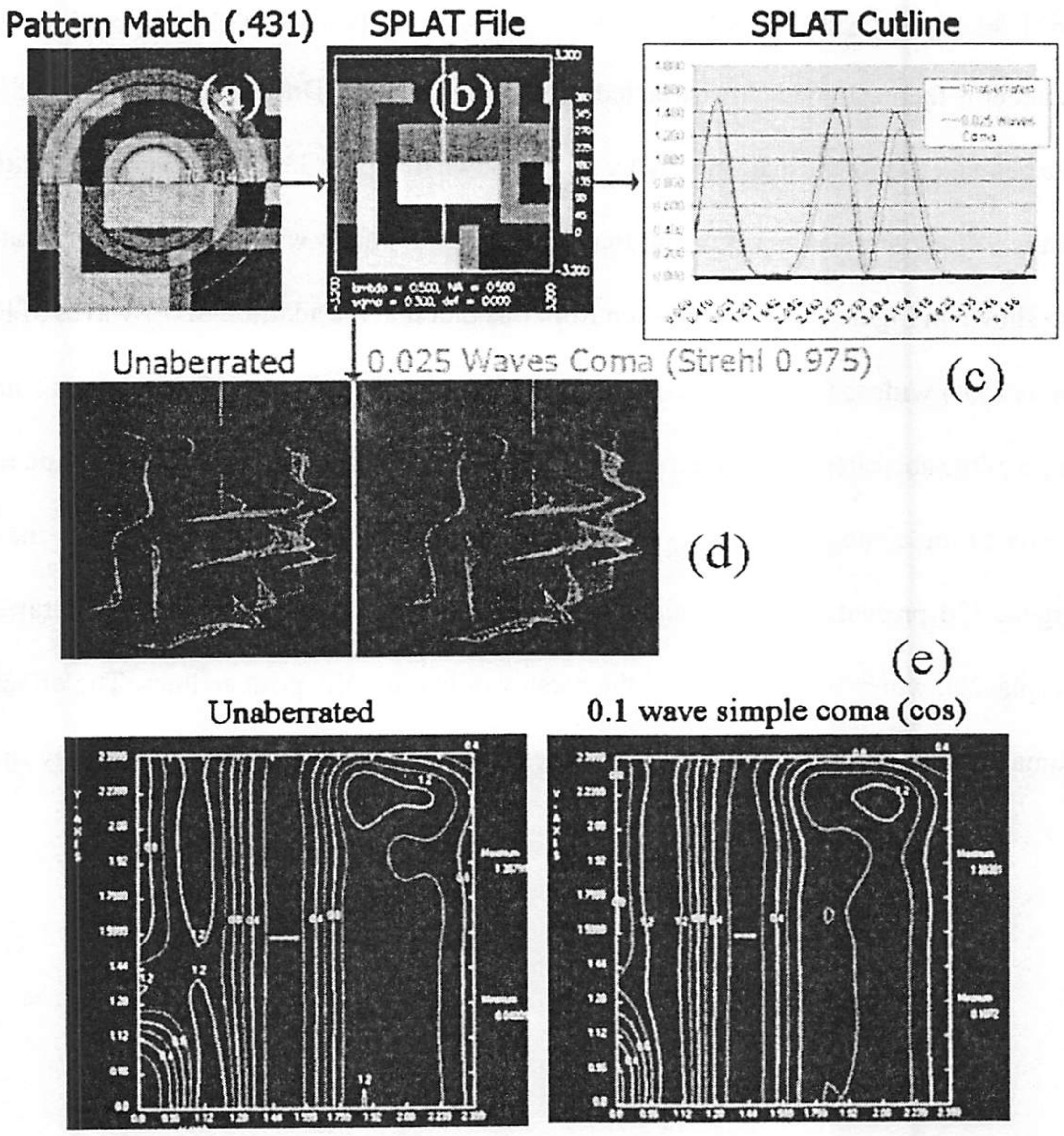


Figure 33: Pattern matcher to SPLAT graphical flow: (a) the match location is (b) extracted to SPLAT for aerial image simulation, comparison of (c) cutlines and (d) 3D contour plots taken with Display3D with and without 0.025 waves of RMS coma and (e) contour plots with and without 0.1 waves of coma.

8.3 Example Matching Runs

Figure 34 illustrates a handcrafted 0/180 PSM layout of test structures designed to be sensitive to the trefoil, coma, and spherical aberrations. This layout and a similar binary version have been designed in Cadence to investigate the qualitative accuracy of the match locations found. This layout consists of arrays of patterns of varied dimensions that are common in real layouts and likely to have a high degree of similarity with the aberration patterns indicated. It is important to note that the actual feature sizes and the optical parameters such as partial coherence play an important rule in determining the match factor, so only the shapes and sizes that resonate with these requirements result in a maximal score. Even though the test layout includes an array of feature sizes, it was not easy to achieve high match factors with the correct match between aberrations and test patterns.

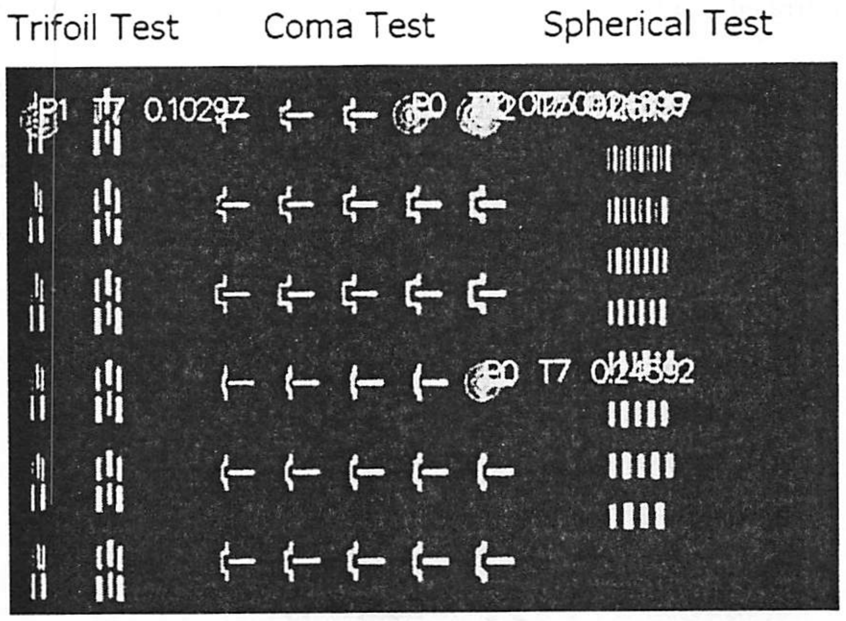


Figure 34: Handcrafted 0/180 PSM test layout for trefoil, coma, and spherical aberrations

The normalized match factor for these test shapes ranges from 0.362 for coma, 0.419 for trefoil and 0.470 for spherical. The match factors of these test structures are relatively low compared to those achieved with more complex geometries. However, the test structures were designed to represent common layout geometries without much complexity. Small OPC features added to these shapes could lead to higher sensitivity since they have a higher probability of resembling the circular aberration pattern shapes. In addition, including both 0-

and 180-degree phase regions in the test structures while keeping them simple increases the match factor by approximately 50% over binary test layouts. A test structure exactly matching the aberration pattern of interest can of course achieve a match factor of 1.0.

A screenshot of a 0/180 industry PSM layout of an interconnect layer taken from [21] is shown in Figure 35. Not only does the pattern matcher locate areas in the mask layout susceptible to aberration effects, but Figure 14 also demonstrates that it can also locate the points along the edges of the mask regions with the highest light intensity by using the IFT of the unaberrated pupil function to predict optical proximity effects. All locations marked in Figure 35 are inside corners, and the match scores and other identifying text symbols are displayed next to their respective matched patterns. The center area of the unaberrated pattern has the highest weighted pixels and thus contributes the most to the match factor.

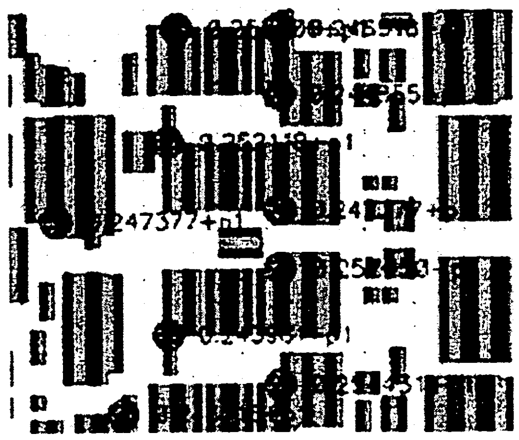


Figure 35: Example of industry 0/180 PSM used for pattern matching. The match locations indicate locations where optical proximity effects are high.

Figure 36 is an example of the coma (cos) pattern matched on a simple binary mask layout with a few features similar to the layout shown in Figure 44. The highest scoring location is on the corner of the inner feature and has a match factor of 0.32. The left central lima bean-shaped area of the pattern, which contains about 40% of the total weight of the pattern, lies on top of the vertical line segment, making a large contribution to the match factor. The effect of coma is to spill light from the vertical line segment into the region around the matched corner, which shifts the corner position to the left or the right.

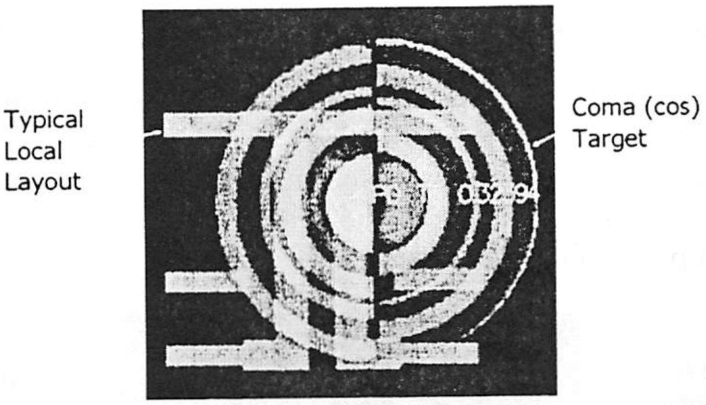


Figure 36: Coma (cos) match on a clear field binary mask

Several patterns can be matched over a layout in a single matching run, such as in Figure 37. This image is a zoomed-in section of the dense layout in Figure 39 where coma, high-order coma, and trefoil aberration patterns have been matched to inside and outside corners in the layout. The match factors here range between 0.16 and 0.3, corresponding to 16%-30% of the worst-case sensitivity to aberrations.

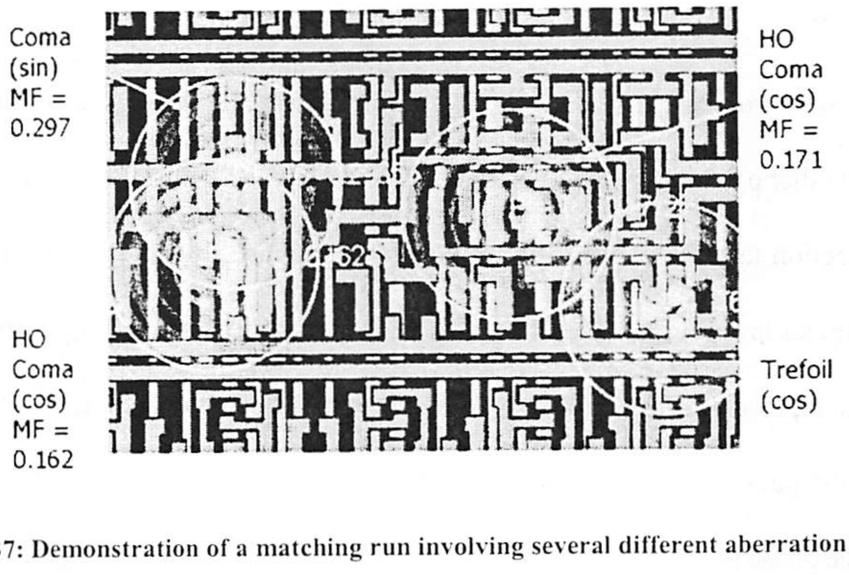


Figure 37: Demonstration of a matching run involving several different aberration patterns

Figure 38 demonstrates the results of combining several basic aberrations and running the pattern matcher on these custom patterns. This layout was created by hand to test the correctness of the matching algorithm and includes 0- and 180-degree phase polygons. These match factors are very low, possibly due to a cancellation effect resulting from the combination of both even and odd aberrations into a single pattern. This leads to complex-valued pattern pixels, and only the real or the imaginary parts of the match factor are kept depending on the phase of the layout shape at the match location.

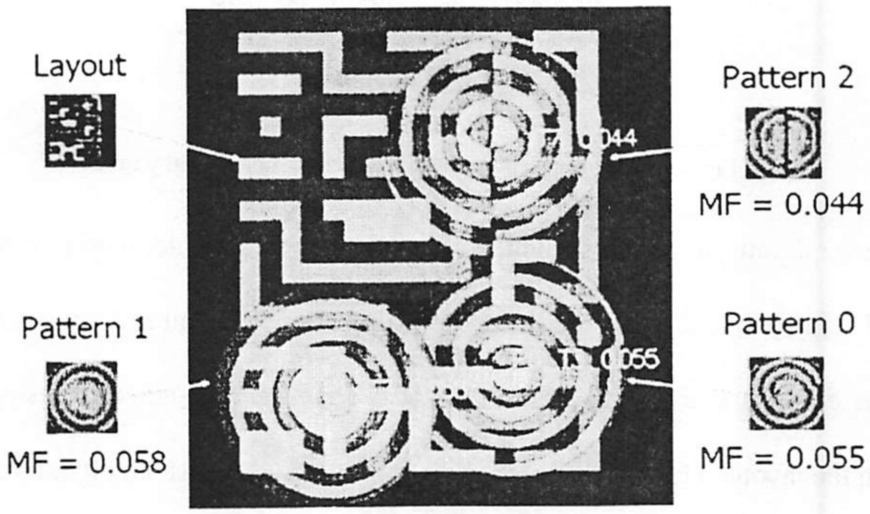


Figure 38: Pattern matching run using custom aberration patterns that are a combination of multiple Zernike terms

Results from larger sections of layouts are also available, such as in Figure 39, where trefoil and other patterns are matched over a dense Field-Programmable Gate Array (FPGA) interconnection fabric. A 140x170μm "fake" 0/180 phase shift mask was created from the first two metal layers of a piece of tiled FPGA interconnect for testing purposes. The blue layout area represents 0-degree phase, red represents 180-degree phase, and black is chrome. Green in the pattern matches to 0-degree phase polygons in the layout and red matches to 180-degree phase polygons. This hierarchical layout contains repeated geometry that results in the same match factor. Therefore, it is sufficient to test each unique cell only once and combine the results together to avoid extra work, but this system has not been implemented due to the complexity of handling overlaps between cells. The image must be zoomed in to see the actual geometry producing the highest trefoil match factor of 0.169, but the text is still readable from the zoomed out view and the match locations are marked with crosses. Coma match factors are as high as 0.34 on this FPGA layout.

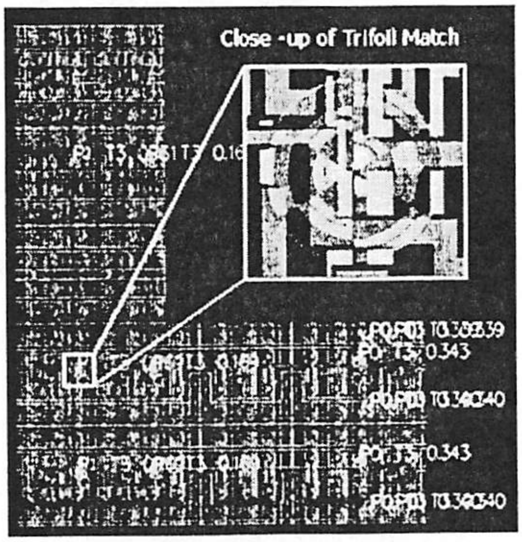


Figure 39: Trefoil pattern matched over a 0/180 PSM section of "imitation" FPGA layout

Shown in Figure 40 is the geometry that is sensitive to the coma aberration with a match factor of 0.381 in the Abacus layout, a complex two-layer mask designed in a 0.5μm technology. The full chip view was shown in Figure 2 of the introduction. This layout includes many 45-degree edges, demonstrating how the pattern matcher can handle non-Manhattan geometry.

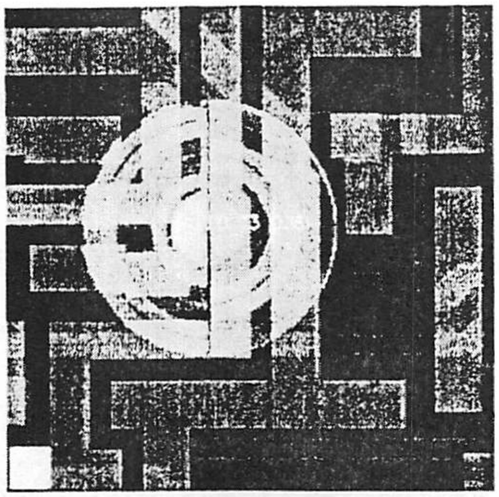


Figure 40: Abacus layout matching run on 45-degree polygon edges

Figure 41 demonstrates that the pattern matcher can handle the reporting of thousands of match locations with little more effort than determining the first location, providing "1000 matches for the price of 1". This is a 3mm by 4mm microprocessor layout with eleven layers and eight levels of hierarchy. The pattern matcher has been run on much larger full chip layouts with tens of millions of transistors, with results similar to the layouts shown.

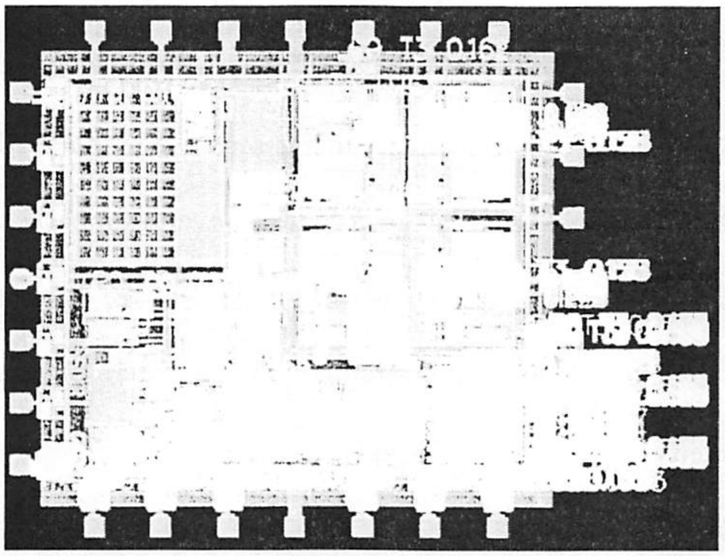


Figure 41: One thousand matches for the runtime "price" of one

8.4 Importance of Aberrations

The first task in verifying the pattern matcher theory is to determine the importance of aberration effects on the printed image through simulations of test masks. The effects of aberrations are described in the following sections. Additional details can be found in [59].

8.4.1 Effects of Aberrations and Optical Proximity

The binary layout involved in these tests consists of a minimum feature (0.6λ/NA) sized horizontal line and a neighboring vertical line of variable width, as shown in Figure 42a. The point of interest was chosen to be the left end of the horizontal line, with the cutline running from left to right. In order to observe the magnitude of line-end shift due to OPC, SPLAT simulations were run with the left edge of the vertical line at various positions leading to a line width (L) of up to 1.2λ/NA and compared to the simulation results of an isolated horizontal line (L = 0). The effect of aberrations was determined by adding 0.025 waves of RMS aberration to the simulation for each value of L and determining the line-end shift due to each aberration.

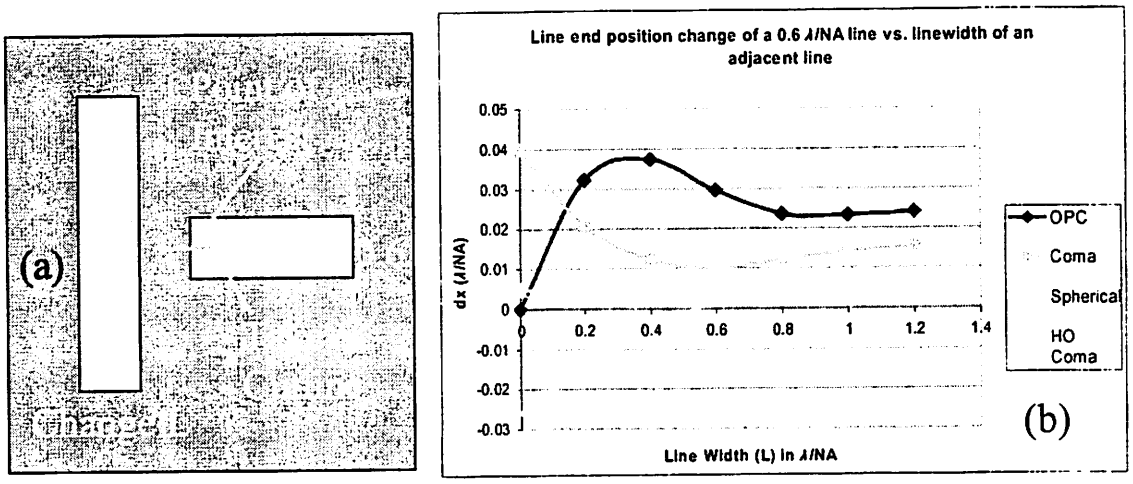


Figure 42: (a) Binary test layout for measuring aberration and optical proximity effects and (b) simulated line edge shift due to aberrations and optical proximity effects on this test layout

A plot of simulated line-end shift of the horizontal line vs. change in width of the vertical line as a result of both optical proximity and 0.025 waves of RMS aberration is given in Figure 42b. The curves are highly variable in the sub-printable linewidth region where L is less than 0.6 λ/NA, but as the line widens the curves approach a constant value. The line-end shift caused by coma (cos), the aberration that this layout geometry is most sensitive to, is about 0.015 λ/NA, which is over half the line shift caused by optical proximity effects due to the vertical line. This illustrates that the effect of aberrations in a good lens with a Strehl ratio of 0.975 is about half that of adding an adjacent shape in close proximity to the line end of interest, or half the value of the optical proximity effect. The effects of aberrations should thus be taken into account during mask layout design. If the aberrations present in a given exposure tool or family of exposure tools are known, then this information could be used to counteract the effect of the aberrations in sensitive areas of the mask geometry.

Figure 43 demonstrates the sensitivity of the match factors of several different aberrations to the width of the vertical line. It can be seen that the match factors are very sensitive to a change in linewidth when the line is small, but as the left edge moves further from the point of interest the match factors stabilize to constant values. This is because the contribution of that area of the line to the match factor diminishes as a result of the fall-off of pattern matrix magnitude and the effect of partial coherence σ. The match factor is near zero for coma (sin) because the layout is symmetric about the x-axis and has zero match factor.

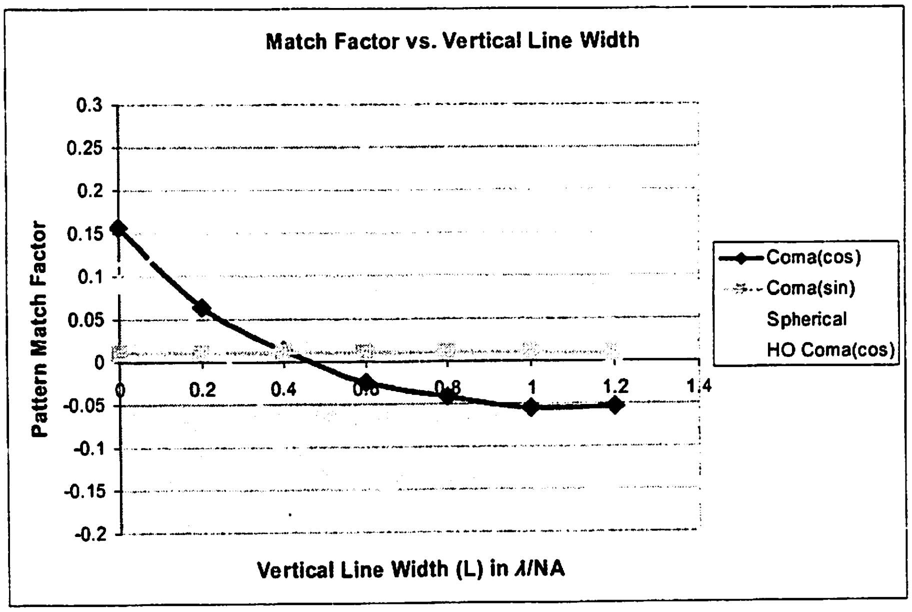


Figure 43: Match factor sensitivity of vertical line width L in Figure 42a

Figure 44a shows a small piece of a clear field binary mask layout and Figure 44b shows a SEM of a similar layout printed at 193nm, taken from [21][18]. The most critical feature in this layout is the small gap between the two vertical lines. The reason that the large vertical lines bulge toward the small horizontal lines is actually unknown; it may be due to optical proximity effects, aberrations, or some other processing issue. This layout was demonstrated to have a coma match factor of 0.324 in Figure 36, but it is not clear that coma is the source of the problems in the SEM. The pattern matching software allows the potential effects of lens aberrations to be tested so that locations like this can be found and repaired before the wafer is printed.

18. SEM courtesy of photolithography section of M. Hanratty of Texas Instruments.

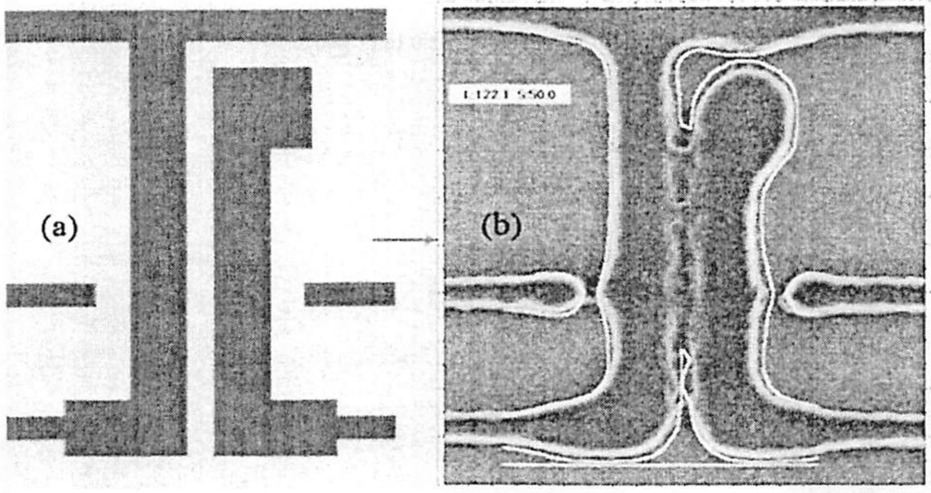


Figure 44: (a) Clear field binary mask and (b) similar SEM image of printed wafer

8.4.2 Line Shift for Dependence on Aberration Level

Experiments were also run to determine how various amounts of an aberration lead to line-edge shift. These tests were performed on a vertical 0.6 λ/NA line/1.0 λ/NA space layout as shown in Figure 45a, with the center of the vertical edge of one of the lines as the point of interest. These layout dimensions were chosen so as to optimize the match factor of the coma (cos) pattern. The amounts of each of the four aberrations were varied from zero to 0.05 waves RMS. Figure 45b shows the line-edge shift resulting from the addition of each aberration independently. Since the layout is symmetric about the x-axis, the effect of coma (sin) was insignificant. However, coma (cos) and high-order coma (cos) aberrations both had a significant effect on the layout. At 0.025 waves RMS aberration, the line-edge shift caused by coma (cos) is nearly 7% of the feature size. The line-edge shift due to coma (cos) is proportional to the amount of aberration, and the same is true for high-order coma (cos) up to about 0.025 waves RMS. However, the curve for spherical, an even aberration, appears to be at a much lower level and quadratic due to the electric fields adding in quadrature at the edge of interest.

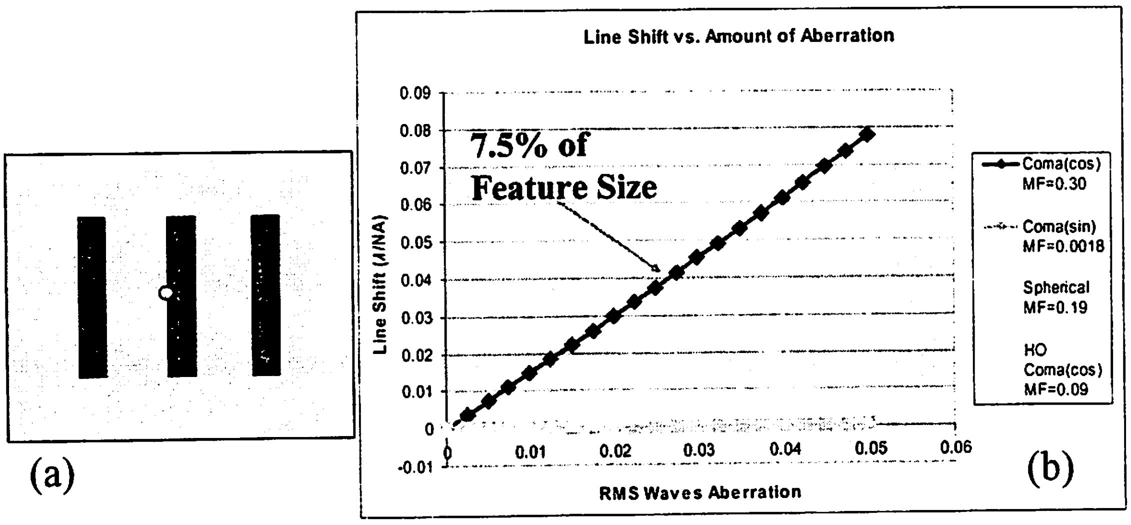


Figure 45: (a) Test pattern and (b) corresponding line edge shift due to various lens aberrations on this pattern

Overall, even aberrations were found to cause large feature shifts in 90/270 degree layouts and very little change in 0/180 degree layouts due to the electric fields adding in quadrature. Odd aberrations showed the reverse situation, with the highest line-edge shift resulting in 0/180 degree layouts.

8.4.3 Comparison of Mask Type and Geometry Type

A set of pattern matches and SPLAT simulations was run on the coma (cos), coma (sin), HO coma (cos), and spherical aberrations for four different mask layout types shown in Figure 46. The test layouts consist of:

(a) An edge on a binary mask,
(b) A line end on a binary mask,
(c) A line end on a phase-shift mask (PSM), and
(d) An edge on a phase-edge mask (PEM).

The points of interest, indicated by the black circles on Figure 46, correspond to the centers of the simulation cutlines and the center match locations of the pattern matcher. The differences in the results for cases (a) and (b) are partially a result of the different profiles of edges versus line ends. The differences between (b) and (c) are purely due to the 180-degree phase region in (c) since otherwise the geometries are identical. Various parameters resulting from the matching runs and SPLAT simulations are presented in Table 5, listed in the same order as in Figure 46. The aberrations resulting in the largest absolute match factor in the pattern matcher are shown below the MF row, and the aberrations producing the largest simulated intensity change and line edge/end shift are recorded in the last row.

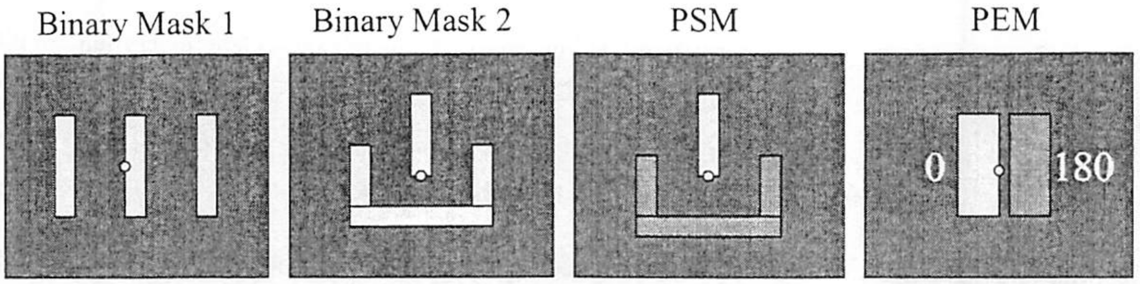


Figure 46: Four simple test masks corresponding to the four entries in Table 5

Parameter	**Binary 1**	**Binary 2**	**PSM**	**PEM**
slope ($\Delta I/\Delta X$)	2.88	1.46	1.33	-3.02
MF @ center	0.19	0.27	0.31	0.56
Aberration (MF)	Spherical	Spherical	Coma(sin)	Coma(cos)
ΔI (intensity)	0.011	0.063	0.12	0.28
ΔX (line shift)	0.0043	0.05	0.084	0.081
Aberration (delta)	HO Coma(cos)	Coma(sin)	Coma(sin)	Coma(cos)

Table 5: Comparison of aberration effects by mask type for binary, phase shift (PSM), and phase edge (PEM) masks

It was found that PSMs show about twice the feature shift as binary masks for a given aberration level, and that PEMs result in a slightly larger intensity change but smaller edge shift due to a higher $\Delta I/\Delta X$ slope. Match factors were highest for PEMs and lowest for binary masks using the same normalization. As expected, edges produce a larger $\Delta I/\Delta X$ slope than do line ends, and the PEM has the highest slope. Therefore, aberrations may play a larger role in the future as more complex PSMs and PEMs with reduced dimensions are created.

The aberration having the greatest effect depends on the layout, and the aberration with the highest match factor was not always the same aberration that led to the highest ΔI or ΔX. As mentioned earlier, even aberrations such as spherical result in only a small intensity and line-edge shift. However, as can be seen in the two binary mask examples (a) and (b) of Table 5, the spherical aberration can still have a large match factor when one of the rings in the pattern is closely aligned with the underlying layout geometry.

8.5 Validation of the Aberration Pattern Matcher

A number of experiments were performed to verify that the pattern matcher does indeed predict the locations in a layout that are sensitive to lens aberration effects. These experiments were performed on a 0/180 degree Phase Shift Mask (PSM) that was derived from the FPGA interconnect layout discussed in Section 8.3. The aberration chosen for these tests was coma in the cosine direction, an odd aberration that is known to cause problems on 0/180 PSMs. The pattern matcher theory has only been verified to be valid on 0/90/180/270 phased mask layouts with purely even or purely odd aberrations. The highest match factors have been verified through comparison with the bitmap algorithm results at those locations to ensure the rectangle algorithm produced correct match factor results.

Figure 47 is a plot of the simulated electric field change due to the coma aberration as a function of predicted match factor. Since SPLAT calculates image intensity and the pattern matcher predicts changes in electric field, the square root of the SPLAT image intensity was taken to get the electric fields used for comparison. As can be seen in the plot, the match factor is a good predictor of the actual electric field change, with an R^2 value of 0.9831. The slope of the curve is an aberration dependent "sensitivity" parameter that is determined through simulation and the amount of aberration. The outlier in the plot, the point with a small match factor but large decrease in electric field, has several potential causes. It may be due to an error in the pattern matcher, though this is unlikely because an identical match factor was calculated with two completely different algorithms. The outlier may also be due to numerical errors in the SPLAT aerial image simulator. Finally, the outlier might be a result of invalid assumptions made about the electric field components as described in the pattern matcher theory of Section 3.1.

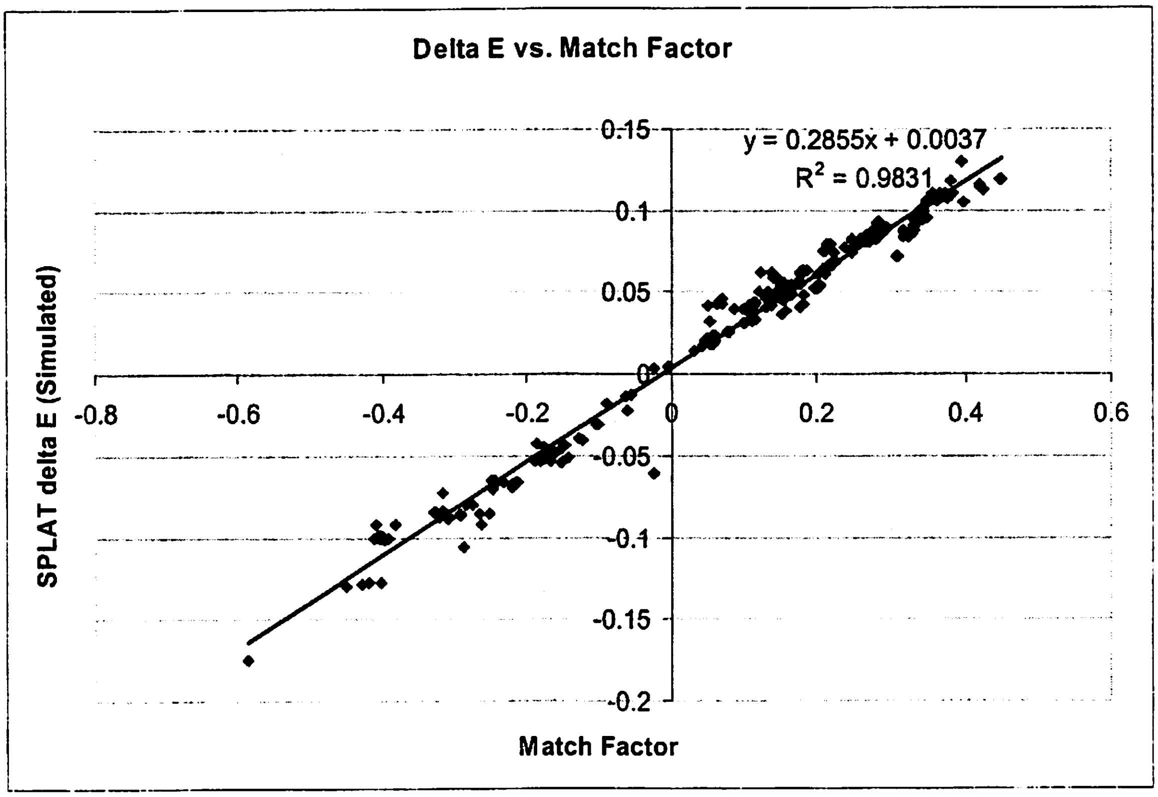


Figure 47: Comparison of match factor prediction of electric field change with SPLAT simulations

This verification method not only proves the validity of the pattern matching theory for the coma aberration, but also helped to debug both the pattern matcher and SPLAT. It has been difficult to verify that the new pattern matching algorithms are correct in all cases, especially for large layouts where comparing the rectangle algorithm results to the original bitmap algorithm results requires prohibitive runtimes. Several bugs regarding SPLAT integration routines and symmetry assumptions have also been found.

Additional pattern matching results with sizeable match factors were verified through SPLAT intensity simulations for a diverse collection of binary and 0/180 PSM layouts. These layouts include simple line and space patterns, a line end with a surrounding feature with and without phase shifting, a phase-edge mask, and variations of the simple binary proximity effect mask in Figure 42. These experiments included batch simulation runs of coma (cos), coma(sin), high order coma, and spherical aberration patterns. In general, odd aberrations such as coma showed an approximately linear relationship between predicted and actual intensity changes, while even aberrations such as spherical showed little simulated intensity changes due to the electric fields of the intensity spillover adding in quadrature. This effect is expected based on the complex number math and pattern matcher theory.

Potential sources of error include the approximation of a finite pattern radius, pixel discretization errors, and layout/pattern sub-pixel alignment errors. Increasing the pattern radius decreases the overall error but greatly increases the runtime and memory requirements of the matching algorithm. It has been observed that increasing the radius of the pattern tends to slightly decrease the values of large match factors and condense the results into a smaller range. The discretization and alignment errors are also an issue because a fixed grid in λ/NA is used to represent a layout drawn in microns, where layout edges that do not correspond to an exact grid line are rounded to the closest grid value.

8.6 Performance

8.6.1 Algorithm Performance Comparison

In most of the timing tests, a 1GHz Pentium III computer with 512MB of RAM was used. Table 6 provides runtime and memory results for the bitmap algorithm with one level of compression, the edge-intersection algorithm, and the initial implementation of the rectangle algorithm before several final optimizations. The newest rectangle and triangle algorithm is 20% to 50% faster than the initial version of the rectangle algorithm shown in the table, and includes data compression that drastically reduces the memory requirements to a fraction of the shown values. The match type column lists the geometry filtering types used in pattern matching, corresponding to edges (EG), line ends (LE), inside corners (IC), and outside corners (OC).

Layout	Area	Rects	Density	Layers	Match Type	Bitmap Time	Edge-Int Time	Rect Time	Edge Mem (MB)	Rect Mem (MB)
fpga_v2	1.0E+07	3300	3.3	2	IC,OC	25.3s	3s	0.44s	27	3
fpga_v2	1.0E+07	3300	3.3	2	EG,LE	183s	8s	0.79s	27	3
abacus	9.6E+08	400K	4.2	14	IC,OC	hours?	126s	23s	50	13
abacus	9.6E+08	400K	4.2	14	LE,IC,OC	hours?	332s	29s	50	13
abacus	9.6E+08	400K	4.2	14	EG,LE,IC, OC	hours?	25 min	174s	50	13
sovachip	3.5E+08	8.5M	240	31	IC,OC	days?	1.4 hrs	68min	220	129
active_test	4.1E+10	35M	8.6	1	LE,IC,OC	weeks?	4.8 hrs	40min	250	149
active_test	4.1E+10	35M	8.6	1	EG,LE,IC, OC	months	8.5 hrs	53min	250	149

Table 6: Pattern matcher bitmap, edge, and rectangle algorithm runtime comparison

Figure 48 shows a log-log comparison of the runtime of the bitmap, edge, rectangle, and triangle pattern matching algorithms. The y-axis in this plot is the total pattern matcher runtime in seconds. The x-axis is the "layout complexity". Since the various pattern matching algorithm runtimes depend on different parameters and statistics of the layout, there is no general metric to use in predicting the runtime of the pattern matcher. The layout complexity has been defined as the number of total rectangles in the flat layout times the average number of match points per rectangle times the density, or the average number of rectangles overlapping a pattern. Each of the pattern matching algorithms increases in runtime with increased layout complexity.

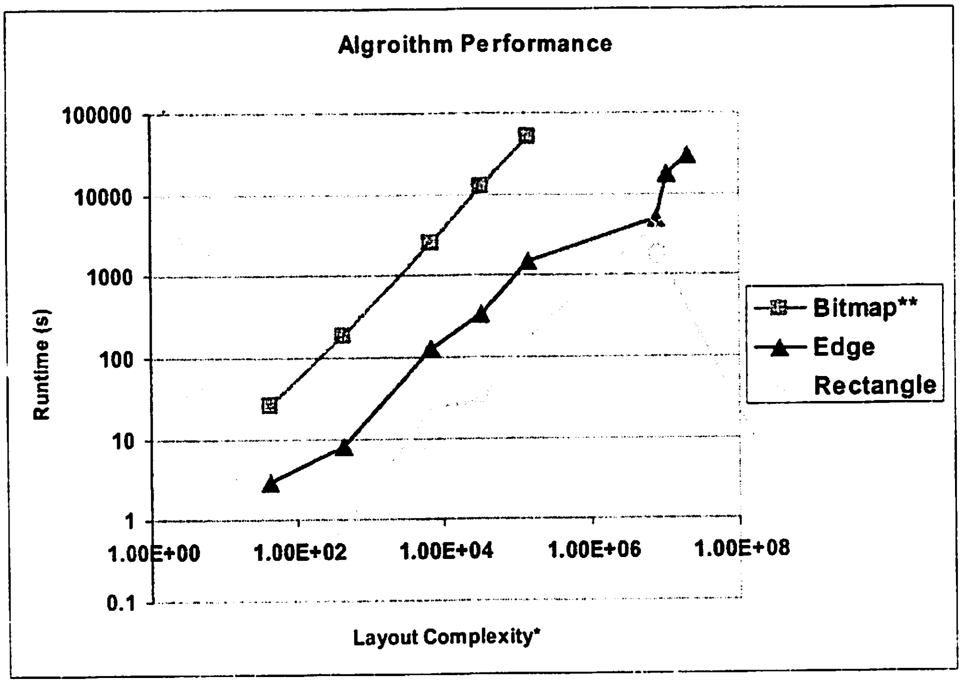


Figure 48: Pattern matching algorithm runtime comparison[19],[20]

As seen in Figure 48, the bitmap algorithm is very slow and cannot even process a larger layout in a reasonable amount of time. The most complex layout, active_test, is

19. * #rectangles*the match_points_per_rectangle*density

20. **larger bitmap algorithm numbers extrapolated

expected to take on the order of a year of processing if using the bitmap algorithm. The edge algorithm is consistently one to two orders of magnitude faster than the bitmap algorithm. The rectangle algorithm is an additional order of magnitude faster than the edge algorithm for Manhattan layouts. However, the rectangle algorithm is approximately the same speed as the edge algorithm on the 3rd complex layout because it contains 45-degree diagonal edges. The rectangle algorithm is inefficient for processing this layout due to the large number of rectangles needed to approximate the diagonal edges. Conversely, the edge algorithm is faster than expected on this layout because the layout contains a large number of overlapped layers that share common edges. If the rectangle and triangle algorithm is used instead, then the diagonal edges are split into a small number of triangles, and the runtime is reduced by about a factor of 3.5 so as to remove the spike in the rectangle algorithm curve.

Several other layouts that contained 45-degree diagonal edges were observed to have a long runtime when the polygons were split into rectangles only. The addition of triangles to the data structure allowed the same geometry to be represented with equivalent or even greater accuracy while using a much smaller number of primitives. Thus, the rectangle and triangle algorithm consistently showed a factor of 2X to 3.5X improvement in runtime over the rectangle only algorithm for non-Manhattan layouts.

8.6.2 Rectangle Algorithm Performance

The rectangle-based matching algorithm was tested on a number of large layouts in various technologies with one or more layers. These layouts were examined for sensitivity to the coma lens aberration and several others. In all cases, the actual match time was proportional to the number of rectangles in the flattened layout, though it was not actually flattened, times the layout density. The density increased with the number of overlapping layers, but in actual mask layouts each layer would represent a different phase and thus would not overlap. Therefore, one layer is sufficient to demonstrate the matching performance.

In the timing tests, a 1GHz Pentium III computer with 512MB of RAM was used. The largest test layout, active_test, consisted of the 234MB GDSII file of the active area mask layout of a microprocessor with area 417mm^2, eleven levels of hierarchy, and 35.3 million rectangles if flattened. The coma test pattern was a 128x128 bitmap with pixel size of 100nm, and the layout contained 2.6 billion potential edge match locations, though only about 20% of these points were tested due to filtering by the internal adaptive match factor prediction algorithm. The pattern matching software took 34 minutes of runtime and about 65MB of memory to read the layout, compute the top 1000 match locations using the rectangle algorithm, and display the results. The performance demonstrated by the pattern matcher is much better than that of OPC algorithms since it is not iterative and does not change the layout geometry.

Performance of the rectangle pattern-matching algorithm was measured on a variety of operating systems and processor/memory architectures. The memory requirements were 10% to 20% higher on 64-bit architectures as opposed to 32-bit architectures due to the increased storage size required for pointers. Table 7 below gives a comparison of active_test runtimes on various systems. The runtime is dominated by the inner loop of the pattern matcher where the actual match factors are computed. This loop contains some floating-point complex number multiplies of pattern weights with layout weights and a large number of memory accesses into the pre-integration matrices, which limit the runtime on the Pentium systems. The integer bounding box tests of the inner loop appear to limit the runtime on the 64-bit systems, which have better memory and floating-point subsystems. The IBM machine spends a great deal of time allocating small amount of memory for rectangles.

Operating System	Processor	Processor Speed	Bit Width	Runtime	Memory
Linux	P4 Xeon	3.2GHz	32	15 min.	65MB
Windows 2000	P4 Xeon	2.0GHz	32	20 min.	65MB
Linux	Itanium2	1.3GHz	64	25 min.	70MB
Windows 2000	Pentium3	1.0GHz	32	34 min.	65MB
IBM AIX	IBM Power3	300MHz	64	150 min.	70MB

Table 7: Rectangle algorithm active_test performance for various systems.

The largest design on which the pattern matcher was run was a critical layer of a post-OPC full chip layout. This 5.6GB hierarchical layout contained over a hundred million rectangles and polygons. The match factor for this pattern-matching run was computed at the corners of the layout for two 128 by 128 pattern orientations, taking 1.3GB of memory and running for 17 minutes on a single 2.8GHz processor. The non-OPC version of this layout took 11 minutes to process. The pattern matcher is clearly much faster than OPC, and several orders of magnitude faster than actual simulation.

8.7 Summary of Results

The results presented in this chapter cover the outcome of architectural choices made in the software development, the physical verification of the aberration pattern matching theory and assumptions, and the quality of the computational algorithms developed for large-scale pattern matching. These results show that the pattern matcher software architecture is capable of directly reading an industry layout and a set of Zernike polynomials and producing a set of match locations with minimal user input. The automatic geometry extraction at result locations and integrated display allowed for easy simulation and verification of aberration effects. The system as a whole is a well-integrated, platform independent pattern matching solution that can easily fit into an integrated circuit design flow.

SPLAT simulation was used to verify the importance of aberrations, which were found to lead to significant line end shift, as much as half that due to optical proximity effects even in good lenses with only 0.025 waves of RMS aberrations. The pattern matcher's prediction of aberration effects on the electric fields at feature edges was found to be in good agreement with SPLAT aerial image simulations. The match factor is thus an excellent indicator of geometries that must be redesigned or otherwise examined in more detail by the designer to assess the negative impacts of lens aberrations.

The performance results indicate that the pattern matching system is ready for use on modern full chip layouts. The procedure of loading the layout, building an efficient hierarchical database, processing the geometry on each layer, filtering out edge and corner locations, and calculating the match factors has been optimized so as to remove all major bottlenecks, leading to a streamlined matching algorithm. The rectangle and triangle algorithm is approximately two orders of magnitude faster than OPC algorithms since the match is performed in a single pass rather than iteratively, a single pattern is used instead of the multiple kernels used in OPC, and the database is built as a static rather than a dynamic data structure. All runtimes, even for large commercial microprocessor layouts, are under an hour and can be executed on a desktop machine with 1GB of memory.

9 Additional Applications in Linking Process and EDA

Predicting the locations sensitive to lens aberrations was an excellent starting application for a pattern matching approach due to the known problems related to aberrations, a solid existing theoretical background, and sufficient previous work to develop the idea. However, the pattern matcher has the potential of matching any set of images to any collection of geometric shapes and is especially adapted to processing the large number of polygons found in integrated circuit layouts. A list of additional pattern matching applications has been generated to determine the features required in the software architecture, such as the ability to perform Boolean and other operations directly on the drawn geometry layers. These applications include longer-range effects which require patterns of a much larger radius then those used for representing aberration targets. Therefore, it was necessary to extend the pattern matching internal algorithms in order to efficiently handle large area patterns. Finally, these applications test the extensibility of the pattern matching theory to different classifications of patterns and multi-layer effects that involve more complex layer weighting schemes. The range of patterns include complex number pixel weightings, pattern equations with infinite poles, non-smooth pixel variation, and patterns whose equations cannot be represented in closed form.

Several examples of Maximal Lateral Test Patterns (MLTPs) as defined in Section 3.5 are shown in Figure 49. The upper left MLTP (a) is one of the original patterns used for determining mask geometries susceptible to residual coma aberration in a lens system. The lower right MLTP is one example of a pattern designed to detect areas sensitive to flare, which potentially has a large area of effect (Section 9.1). The upper middle MLTP (b) depicts combining the statistics of the relative misalignment of two masks as a Gaussian-shaped test pattern (Section 9.2). The upper-right MLTP (c) indicates how know defects from a mask inspection report might be incorporated into imaging effect analysis prior to combining with misalignment to assess soft-error yield reduction (Section 9.2). The lower left MLTP (d) is a donut-shaped pattern that tests for reflective slopes of polysilicon crossing the edge of the active region to determine the relative strength of light reflected laterally into the resist for use in assessing reflective notching effects (Section 9.3). The lower middle MLTP (e) is for modeling the heatsinking due to lateral flow of heat into the active region to avoid the polysilicon melting problem of Laser-Assisted Processing (LAP) (Section 9.4). For Chemical-Mechanical Polishing (CMP) issues such as dishing and erosion and loading in plasma etching, either specialized matching procedures based on geometry fill densities or a very large MLTP can be used (Section 9.5). Some of the material for this chapter was taken from Neureuther and Gennari [60].

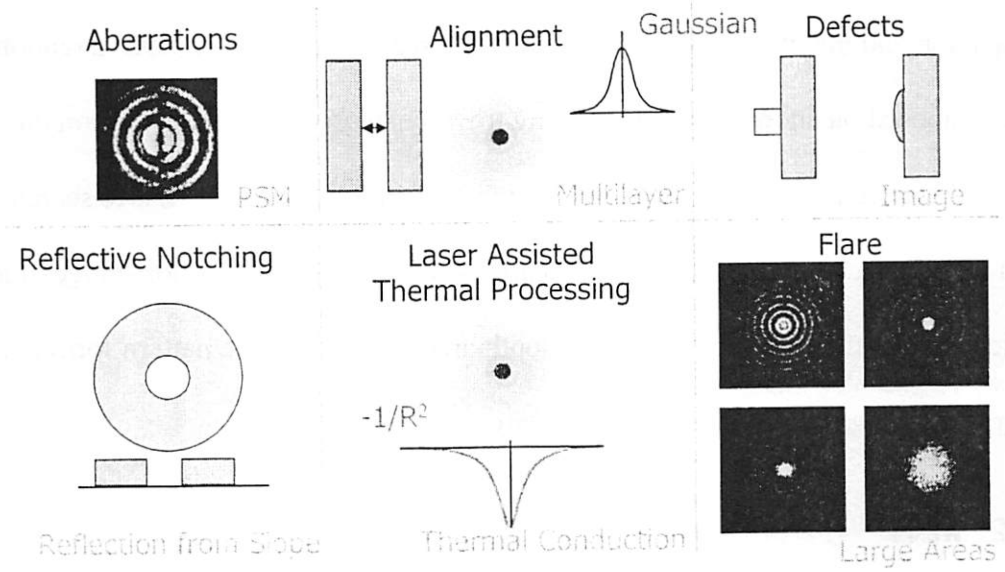


Figure 49: MLTPs for various pattern matching applications

The pattern matching applications discussed in this chapter fall under the idea of

Process Proximity Correction (PPC) [61, 62], which complements OPC in leading to a more manufacturable design. These extensions of the pattern matching idea have been derived from theory and from intuition but have not been formally verified. The pattern matcher has the capability of simultaneously processing multiple mask layers, including Boolean layers, and is therefore well equipped to match this diverse set of patterns involving multilayer effects.

9.1 Flare

9.1.1 Problem Definition

Unwanted scattered light intensity, or flare, affects image-quality and OPC behavior and is becoming an increasing concern in lithography as wavelength decreases [63]. Lai's study [64] shows that scattered light from the projection optics has a large impact on image degradation and that it is possible to consider the scattering phenomena as a random phase screen imposed on an ideal lens. Increasing the robustness against degradation due to flare early in the design stages will help improve manufacturability. The goal is to search through a full-chip layout and quickly identify locations worst impacted by short-range, mid-range and long-range flare effects. A more in-depth analysis of the flare pattern formulation and pattern matching experiments involving flare is given in [65].

9.1.2 MLTP

Flare is an intensity phenomenon and thus requires the use of statistical optics in order to perform the analysis. Goodman [66] has shown that one method of considering flare under partially coherent imaging is a multiplicative Optical Transfer Function (OTF) that scales the image spatial-frequency content. More recently, Lai [64] has shown details on the procedure to obtain the expected value of the OTF for flare from a phase screen representing a set of surface-scattering profiles.

This analysis uses the surface scattering model of Lai, restricted to the case where roughness statistics of each surface are described by a Gaussian, ergodic autocorrelation function:

$$\gamma = \sigma^2 \exp\left(-\frac{r^2}{w^2}\right)$$

Here $\sigma 2$ represents the RMS surface roughness, w represents the spatial coherence radius in microns, and r is the relative distance between two points of interest on the scattering surface. The actual equation used to generate the flare MLTP in the pattern generator is derived from the following intensity sensitivity function:

$$PSF_{I,e} = FT^{-1}[OTF_{I,o} e^{-\sigma^2} (e^{+\sigma^2 FT(\gamma)} - e^{+\sigma^2})]$$

PSF_E can then be determined by taking the square root of PSF_I. The derivation of these equations and determination of the MLTP is provided in Sections 2.2 and 2.3 of [65].

Modifications were made to the pattern generation software so as to generate patterns from a weighted linear combination of flare functions with different values of σ and w. Batch capability and exporting to images and SPLAT files were added to explore the pattern dependence on the flare parameters. Figure 50 shows the output pattern images from a single pattern generator input file run with a range of values for σ and w. The alpha (transparency) component of the pattern color displays the magnitude of the pattern value ranging from 100% to 0.1% of the normalized peak value in log scale. Blue indicates a zero-degree phase, while yellow indicates a 180-degree phase. The patterns in the upper left with small flare amplitude and short coherence length show concentric rings that slowly fall to zero at large radius values, where the zero crossings are similar to those in the Airy function. As w is increased, the outer rings are attenuated and the patterns transition to an increasingly wider Gaussian function as expected. The outer rings have completely disappeared at w= 1.7, leaving only a large Gaussian that eventually transitions to a constant DC flare value for large σ and w. Note that the lower right image has been clipped to a circular area and in fact the blue colored area continues out to a large radius.

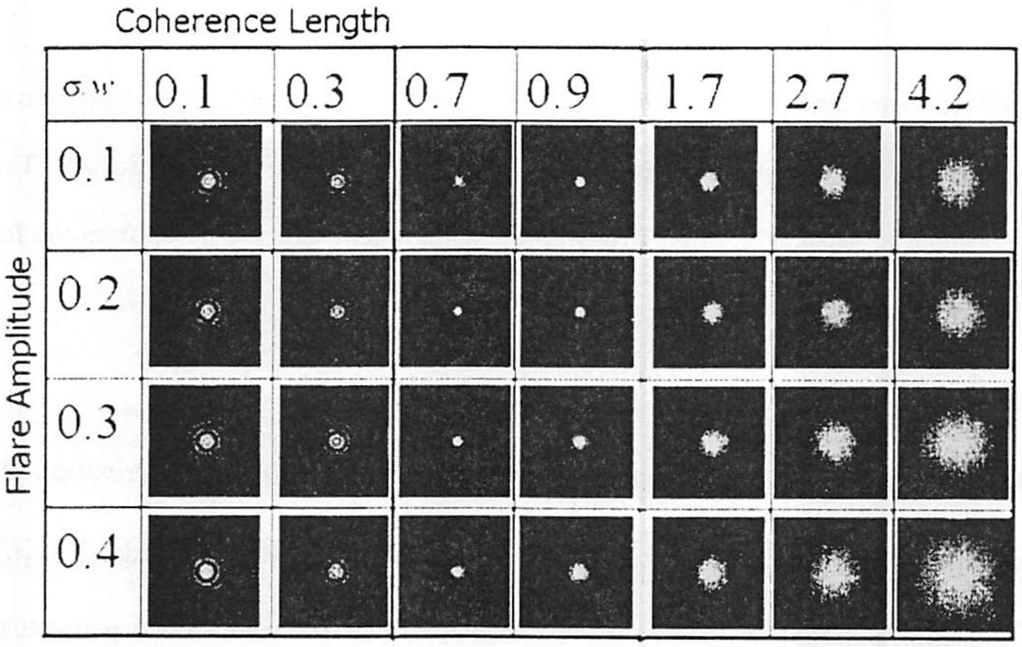


Figure 50: Flare MLTPs for various values of σ and w

9.1.3 Flare Results

The pattern matching results show good agreement in predicting flare sensitivity for several test layouts as described in [65]. The line and blocker test [64] consists of two exposures on a clear field mask. The A set of thin lines is exposed at normal dose, and the flare intensity is extracted from the change in linewidth between the two steps. A set of wider blockers are exposed at a much higher dose (10X) in order for the flare to spill a large amount of extra light under the blockers. Larger blockers increase the distance light must scatter in order to affect the linewidth, thus the CD change decreases with increasing blocker size. Lines under the smallest blockers were expected to have the highest match factor. Patterns with area spread out as opposed to concentrated at the center are expected to have higher match factors since the blockers eliminate the pattern contribution to the match factor in that region. The pattern matcher was used to predict the sensitivity of flare for each blocker width and flare patterns with several values of w, and the match factor results were as expected. Additional tests included a five-bar knife-edge pattern and a square box test [67].

9.2 *Misalignment and Defects*

The pattern matcher can be used to provide a fast evaluation of the robustness of a manufacturing process, as illustrated in this simplified ASIC process flow. The example is based on discussions and SEMs provided by LSI Logic [68][21] and presented in [60]. As each new mask layer is written it is inspected and a long list of defects is reported to the manufacturing engineer. The engineer must then predict the consequences of each defect on yield and decide to accept, reject, or repair the mask based on an assessment of the delay and cost versus yield tradeoff. Information on defects and their printability can be found in [26] and [69] and a discussion of phase defects can be found in [70]. A number of systems exist that handle creation and simulation of defect databases [71], but these systems are complex

21. SEMs provided by Neal Callan and Ebo Croffie of LSI Logic.

and require a considerable amount of time and computation power.

A typical polysilicon gate mask defect found in this scenario is shown in Figure 51a. Here a small region of undesired chrome remains on the edge of the mask. When this mask is used to print an image on a wafer, the defect produces a lateral bulge of polysilicon. As shown in the top and bottom SEMs of Figure 51b and c, the presence of this defect does result in a short of the polysilicon to the metal that interconnects the source and drain contacts. This short is unexpected from examination of only the polysilicon mask without considering layer-to-layer effects involving the metal layer mask. Due to statistical effects in alignment, the occurrence of this short is also statistical in nature and creates a soft error at die sort. The pattern matcher can be used to predict the effects of individual defects on soft yield loss in the presence of both imaging and misalignment effects.

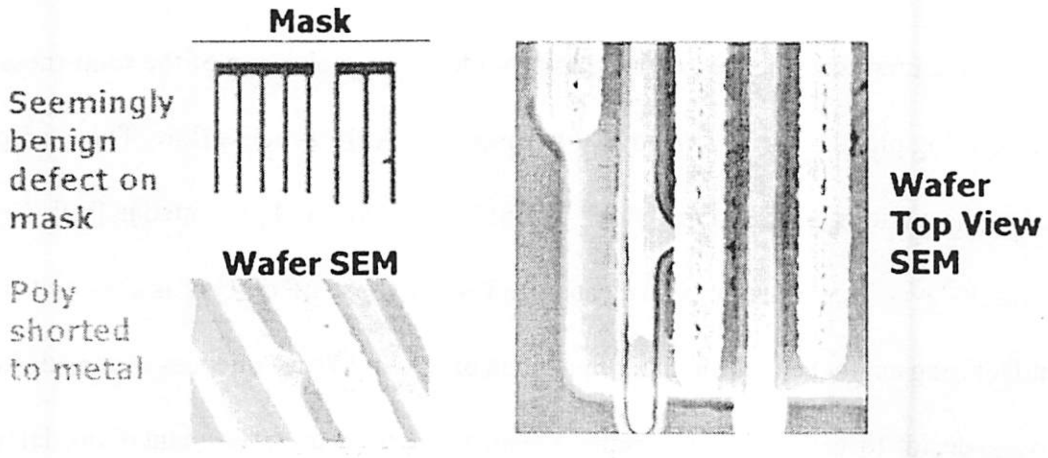


Figure 51: Example of a defect on a polysilicon gate: (a) SEM of defect on poly mask, (b) bottom view SEM showing polysilicon shorted to metal by defect, and (c) top view SEM

Figure 52a shows the polysilicon (POLY), contact (CON), and metal (METL) masks in the layout as drawn by the circuit designer and the associated device cross-section. Figure 52b shows how misalignment and corner rounding due to imaging affect the presence of the defect-induced short. Considering the proximity of the defect on the POLY mask to the edge of the METL mask, it is clear that the alignment statistics will play a significant role, particularly if the alignment of METL and POLY is indirectly via the CON mask. Once the printed image shape has been determined by evaluating the imaging system, the yield reduction can be estimated by incorporating a statistical model for the cumulative misalignment into a MLTP. Though the true values would depend on the statistics of the actual process that are taken from the fab, the MLTP could be described by a generic two-dimensional Gaussian distribution function with standard deviations corresponding to the misalignment distributions in the horizontal and vertical directions. The collective results of these effects are shown in Figure 53. By normalizing the MLTP, the area bounded by the edge of an adjacent feature on an electrically adjacent mask level would give a reasonable estimate of the yield loss due to the presence of the defect. For the current example, the likelihoods of POLY shorting to CON or METL behave differently with misalignment error and defect size primarily due to the larger statistical variation resulting from indirect alignment.

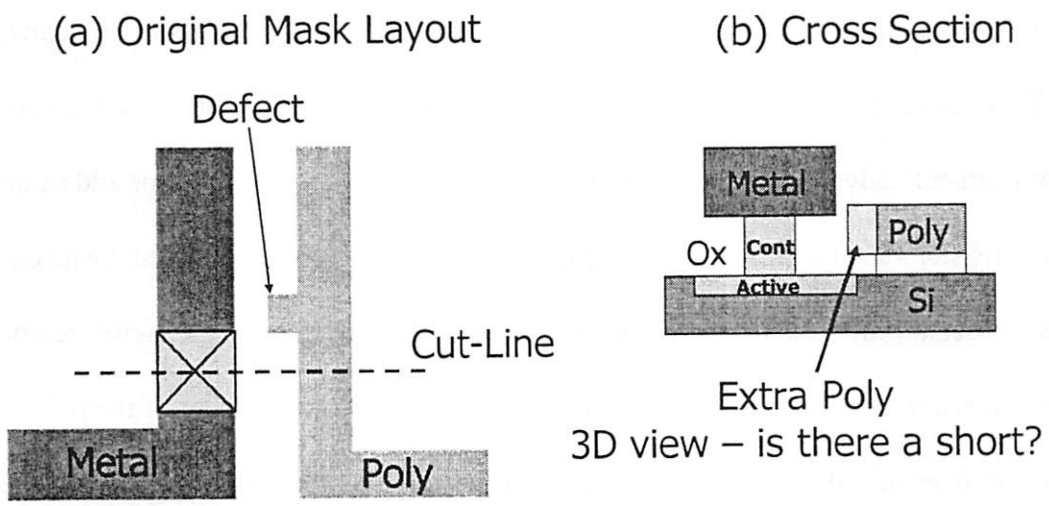


Figure 52: (a) Drawn mask layout with defect and (b) device cross-section showing potential short due to defect

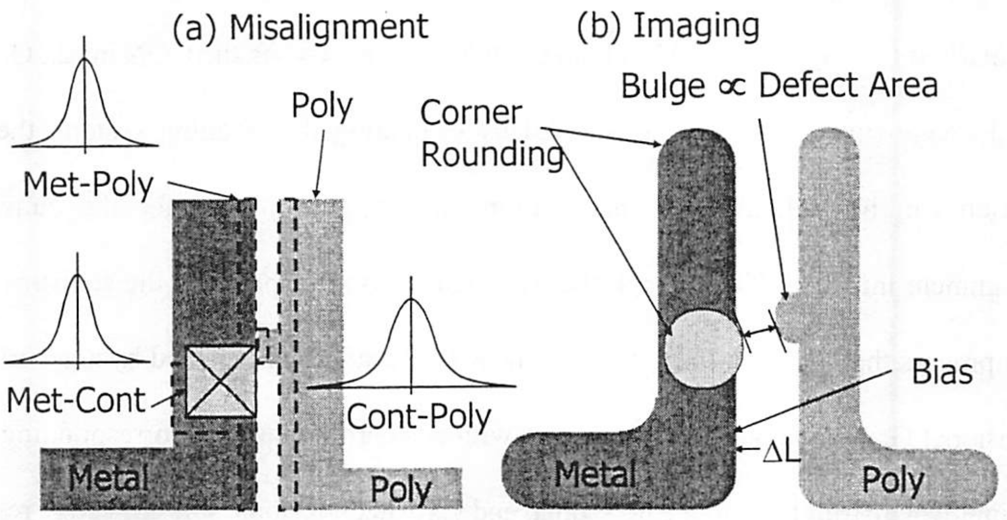


Figure 53: Defect and misalignment problems may cause shorting between polysilicon and metal. The drawn image is statistically affected by (a) misalignment and (b) imaging when determining the effects of defects.

9.3 Reflective Notching

Historically, one undesired lithographic effect has been the lateral scattering of light from the wafer topography into resist that should remain unexposed. This unwanted scattered light tends to produce linewidth narrowing in positive resist know as reflective notching [72, 73]. There are a number of potential solutions to the reflective notching problem such as bottom anti-reflective coatings (BARCs) and dyed resists, but these solutions add additional processing steps and increase manufacturing costs. One situation where reflective notching is known to occur is in the area where a polysilicon gate slopes down into the active region of a MOS transistor, as depicted in Figure 54. The changes in surface height of the polysilicon deposited over the edge of the active area and the reflectivity of the polysilicon together create the lateral reflection. For this process effect, the active area mask (ACTV) along with the process flow could be interpreted as creating the sloped surface, or the "bird's beak", that directs the light laterally into resist that should remain unexposed. Thus for this process effect it is the device-facing edge of the active mask that must be recognized and expanded to the size of the sloped surface to make the estimate. Also, the normally dark region of the unexposed polysilicon on the POLY mask must not be allowed to contribute to the matching since the light will not reflect from there.

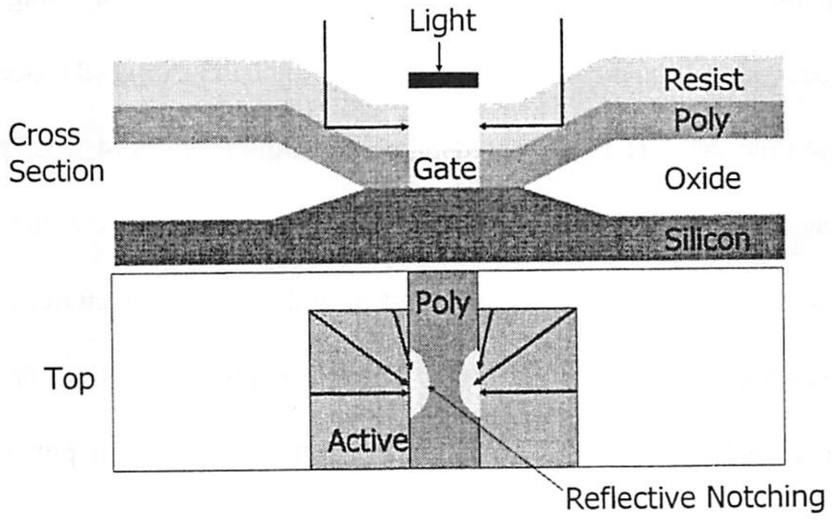


Figure 54: Demonstration of the reflective notching phenomena

The MLTP is thus a donut-shaped ring that acts only on the active region sloped surface derived from ACTV where the POLY layer is not present. Thus the match factor is computed only at points along a polysilicon gate defined by the Boolean layer:

MATCH_LAYER = POLY *AND* ACTV,

and based on the weights of another Boolean layer:

WEIGHTED_LAYER = (*EDGE*(ACTV) *GROWBY* slope_width) *ANDNOT* POLY.

The inner radius of the donut is determined by the minimum spacing between POLY and ACTV, and the outer radius is limited by either the width of the sloped area of the active region or the length at which light is attenuated to a low intensity by the resist. The weights of the pixels within the donut area can be generated by any radial function that gradually falls to zero at the inner and outer radius of the donut and reaches a value of 1.0 in the center.

9.4 Laser-Assisted Thermal Processing

Another semiconductor processing application where pattern matching can be used is in locating areas where polysilicon is likely to melt during Laser-Assisted Processing (LAP) / Laser Thermal Processing (LTP) / Laser Spike Annealing (LSA) [74, 75, 76]. The oxide within the trenches of a trench isolation layout has far lower thermal conductivity than the silicon substrate. The thermal insulating properties of the oxide prevent dissipation of heat built up in the polysilicon as laser light strikes the surface. Polysilicon lines far from the heat sinking active area silicon have a high likelihood of melting, resulting in potential shorts and open circuits. Polysilicon lines overlapping or near the active area silicon are cooled through lateral heat transfer across the surface of the oxide and the flow of heat down to the silicon substrate, so melting is not a problem. To estimate the cooling due to heat transfer to the active area, a negative $1/R^2$ MLTP was used as shown in Figure 55. This MLTP is a simple approximation to the thermal characteristics of laser-assisted processing, but nonetheless provides a good estimate of polysilicon areas that are subject to higher temperatures.

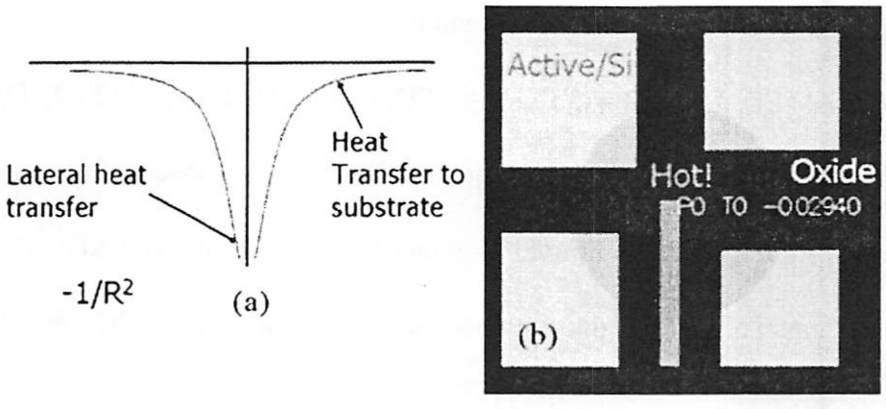


Figure 55: Laser-assisted processing (a) MLTP and (b) demonstration of match location

Example screen shots of a register cell layout tested for heating of polysilicon during laser assisted thermal processing and for reflective notching of a polysilicon gate are shown in Figure 56a and Figure 56b, respectively.

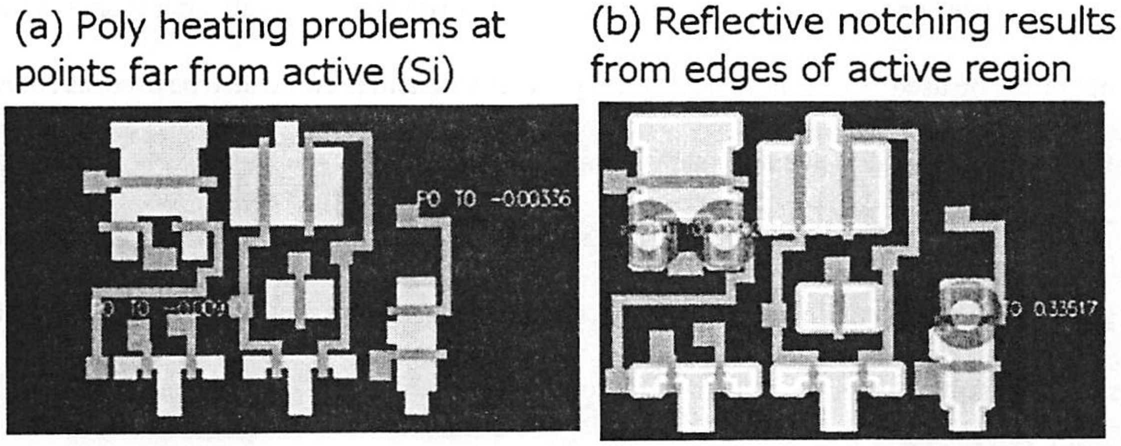

(a) Poly heating problems at points far from active (Si)
(b) Reflective notching results from edges of active region
PO TO -0.00336
O 0.33517

Figure 56: Experimental pattern matching runs on a register cell that locate areas sensitive to (a) polysilicon melting during LAP, and (b) reflective notching

9.5 *CMP Dishing and Erosion and Plasma Etching*

Chemical-mechanical polishing (CMP) and plasma etching [77] can include, in addition to very local pattern dependent effects, pattern density effects that act over a larger scale. The larger, more challenging scale effects include erosion and dishing [78] in CMP and loading in plasma etching. CMP dishing and erosion are complex processes [79] that result in the thinning of metal and dielectric (in the case of erosion) in areas that are far from any harder supporting materials, reducing electrical performance and reliability of the circuit. Plasma etching suffers from reactant loading problems that results in reduced etch rates in areas of excessively exposed substrate surfaces.

For these applications, the pattern would consist of positive real numbers that decrease with radius and model the effects of distance on the processing effect. The mask layers in the layout provide wafer topography information or represent the hardness of various wafer materials that undergo the CMP process. Instead of pattern matching, simple procedures based on finding a point that is the maximum distance from all other shapes in the layout might be used. Some measure of local geometry densities could also be used, as could a measure of perimeters, average shape counts, or sizes.

An example for dishing in CMP is shown in Figure 57, where the point with the largest distance from all other features has been found. Once these match locations are found, areas where polygon density is too high or too low can be altered so as to reduce the match factor. This can be done by implementing CMP filling and slotting algorithms [80] directly in the pattern matcher, though this procedure has not yet been attempted. A recursive rectangle filling procedure was implemented in Cadence with SKILL, but this filling method was inefficient for processing of large areas.

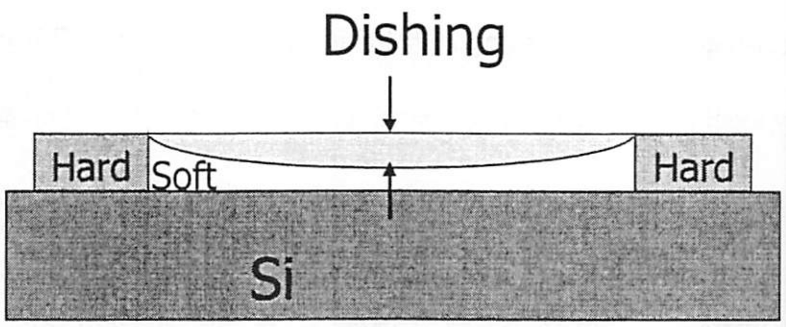


Figure 57: Example of CMP dishing of soft material between two supporting posts of harder material

9.5.1 Large Area Patterns

Pattern matching applications such as CMP dishing and erosion and long-range flare have a much larger area of effect than the original lens aberration application. A test pattern of a scale similar to the physical phenomena may encompass hundreds to thousands of layout features. Therefore, the pattern must be a large number of pixels in width, and the number of rectangles and triangles overlapping the pattern is also much larger that with standard patterns. Since the runtime of the rectangle and triangle algorithms scales with the average number of shapes overlapping the pattern, the overall runtime scales as the pattern width, or radius of the processing effect, squared.

Consider an example of a CMP matching run where the CMP pattern is $100\mu m$ on a side in a $0.1\mu m$ process (so the math is easy). This translates into a pattern that is 1000 features on a side, and with 4 pixels per feature the pattern is 4000 pixels by 4000 pixels. The runtime of this example would be 1000 times that of a 128 by 128 lens aberration pattern. Furthermore, the pattern and pre-integration matrix of the rectangle algorithm alone would require 512MB of memory. Clearly, this large of a pattern will cause performance problems, and something must be done to improve the algorithm.

There are several ways to more efficiently handle large patterns. An FFT-based algorithm could in fact be faster than the rectangle algorithm in this situation, but would still take on the order of days to run on a full chip layout. A larger pixel grid of several minimum feature sizes could be used, but then small shapes less than a pixel on a side would disappear, underestimating the amount of geometry in the area. One alternative is to match only on every nth point in the layout instead of every pixel. It is probably unnecessary to compute the sensitivity to CMP effects at 25nm intervals; testing the layout every 250nm might work just as well. The time per match factor computation would still be large, but testing at a granularity of every tenth pixel would give as much as a speedup of 100 if every pixel was originally tested or a speedup of ten if edges are being tested.

Another method of speeding up large area patterns is by applying grayscaling techniques and adaptively refined the layout grid resolution. This procedure is similar to the one used to compress the layout in the bitmap algorithm, as discussed in Section 6.2.1.The layout is first generated on a low resolution, large pixel sized grid. The pattern is down-sampled and the layout is grayscaled, and match factors are computed on this low-resolution layout. The regions with high match factors are then refined to a higher resolution, and run again. This process is repeated until the highest resolution is reached, in which case there are only a small number of high match factor locations left at this resolution. This algorithm has not yet been written, and the runtime analysis is very complex and dependent on the thresholds and error bounds. It is anticipated that, if implemented along with selective point matching, the pattern matcher can be made to run very quickly even on large pattern sizes.

9.6 Phase Etch Depth Errors

Another potential application of pattern matching is to determine the locations in a circuit design that are impacted the most by errors in phase etch depth during fabrication of a phase-shifting mask (PSM) [81]. The effect of phase errors of nearby mask openings on a central observation point is the spillover of an error term in the electric field that changes the intensity at the observation point. There is no single pattern that represents the worst case geometry sensitive to phase errors, so the pattern matching procedure is slightly different in for this application. The MLTP in this case is the unaberrated point spread function, or the Airy disk, which represents the proximity effect due shapes adjacent to the test point. The match factor would therefore be based on a difference between two pattern-layout correlation calculations: one using polygon layer weights representing the correct mask phases, and another pass using layer weights adjusted for a user-specified amount of phase error. The results of pattern matching will contain the locations where the phase error leads to the largest electric field change at the center location, which are the areas most sensitive to phase errors.

This application actually involves two passes of the correlation computation, which is not currently supported in the code. This would, however, be fairly straightforward to implement assuming the program reads a list of perturbations for each mask phase. In order to speed up the matching in the case where only the electric field change is needed, only layers representing the phases to which the error terms are added need to be considered because the unaffected layer weights will cancel each other out. A single pass could be used to only consider the error terms of the phases, but this would not be sufficient for calculating the intensity change and feature edge shift due to phase errors since the individual electric field terms are needed as described in Section 3.3.

This technique is related to the class of PSM test mask patterns introduced in [81] which are designed to monitor errors in mask making. An example of a pattern designed to monitor phase etch error is shown in Figure 58a. Here, the pattern is constructed with 0- and 180-degree phase regions surrounding a 90-degree central probe. With an equal amount of 0 and 180 regions at any particular radius, the net spillover of electric field to the center is nominally zero for a perfectly constructed mask. However, should an error in mask making occur (i.e. the 180-degree region is inadvertently 185 degrees), then the residual electric field spillover interacts with the probe. Thus, the difference in measured intensity of the probe position compared to a nearby, isolated probe serves as a direct measurement of phase etch error, as shown in Figure 58b.

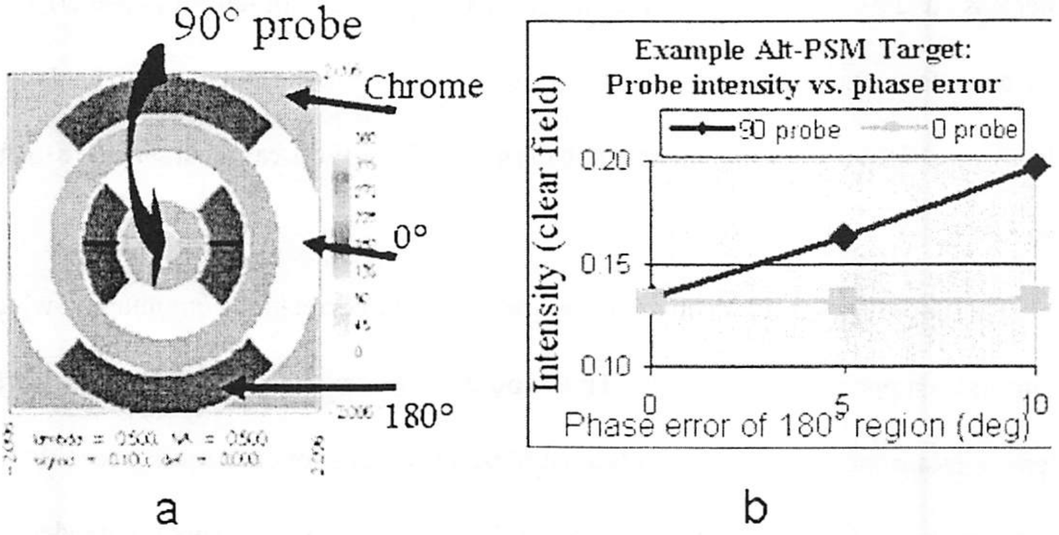


Figure 58: (a) Example mask pattern to detect errors in mask making and (b) The intensity of the probe compared to a nearby isolated probe is a direct measurement of phase etch error of the 180 degree shifted regions

9.7 Other Applications

Pattern matching is extensible to other applications, including applications outside of the realm of integrated circuit lithography and processing. For example, the pattern matcher can possibly be used for handwriting or signature recognition, where the handwritten text is the "layout" and the each character in the matching character library can be a "pattern". In fact, it can be used in any situation where the user is searching for a small image in a much larger set of polygons. Furthermore, since polygons can be represented as images and images can be represented as polygons (each pixel being a rectangle), this pattern-matching algorithm can be extended to a multitude of applications. However, it is not efficient for all applications and also cannot deal with pattern transformations other than translation, mirroring, and 90-degree rotations. Other matching algorithms are required to deal with rotation- and scaling-independent matching.

The layout database used for pattern matching can be useful by itself. The database allows for efficient access and query operations, which are ideal for graphical display and layout export operations. Modules can be added to the pattern matcher for analyzing pattern densities and shape statistics, comparing two layouts for verification purposes, or for computing the sensitivity of transistor gates to plasma damage (antenna effect) [82].

9.8 Summary of Applications

The various pattern matching applications discussed above were important to the evolution of the software tool into a more complete system that was general enough to be used in many areas of integrated circuit processing. A broad collection of Boolean and other layer operations and a complete set of filtering options were necessary for the system to match just about any pattern to almost any collection of polygons. Applications such as CMP, plasma etching, and flare require large pattern sizes and thus specialized pattern matching approaches are needed to efficiently handle these patterns. These types of applications drove the search for better algorithms since the original bitmap algorithm's runtime was so strongly dependent on pattern size.

The pattern matcher is potentially useful for identifying areas in a layout sensitive to a number of residual processing effects as described above, and there are also a large number of potential applications not discussed here. Though these preliminary results are based on simplified models of the underlying processing effects, the pattern matcher can likely be used to accurately locate areas in a layout sensitive to some of these effects using more rigorously derived patterns. It also appears that the system can match these patterns at speeds similar to aberration patterns so that the matching run should still take under an hour on a standard desktop computer. This assumes a solution is found that enables the large-area pattern matching to proceed more efficiently than with the rectangle algorithm, for instance at lower pixel resolutions, at least as an initial filtering step. Applications such as flare can take additional runtime as there may be a large number of potential patterns to match, but there are likely ways to save computed values from one flare pattern and apply the results to similar patterns.

10 Conclusions

The pattern matching software tool described above is capable of identifying areas in a mask layout where residual process effects such a lens aberrations can lead to problems in the printed shapes. This system provides an EDA framework for linking residual process effects from the TCAD area back to the design stage. The efficient rectangle and triangle algorithm allows the pattern matcher to run full-chip much more quickly than OPC algorithms, and it scales well on parallel machines using both MPI and Pthreads. A designer can run the pattern matcher on the raw CIF or GDSII file of a layer of an entire microprocessor layout and receive a list of many thousands of geometries sensitive to the process effects of interest in under an hour on a standard desktop computer.

The effects of lens aberrations have been shown to be important in determining the size and location of printed features, producing as much as half the line edge shift as optical proximity effects. This initial application drove the pattern matching theory and system development. Pattern matching results for lens aberrations were found to be in good agreement with electric field change determined through SPLAT simulation. Once the pattern matcher has found the areas sensitive to aberration effects, the designer can correct the problem by moving shapes so as to reduce the match factor or compensate for the predicted line edge shift. It may also be possible to add a method of automatic layout correction for aberration effects by moving polygon edges similarly to what is done in OPC.

Pattern matching has been shown to be extensible to other areas of integrated circuit manufacturing such as determining the locations most subject to flare, reflective notching, laser-assisted processing, defects and misalignment, CMP dishing and erosion, loading in plasma etching, and mask phase errors. Some of these ideas required additional software features such as the ability to constrain matches to areas where polysilicon overlaps the active region, forming transistors. These applications involve a range of layer Boolean operations such as AND, OR, XOR, ANDNOT, EDGE, and GROWBY and multiple layer weights as well as a wide range of pattern sizes. Large patterns are time-consuming to match on full chip layouts and thus a better method is needed, perhaps one involving pattern matching on a low resolution approximation of the layout and then refining the top matches or only matching at every few grid points. It is expected that pattern matching has the potential to be an efficient and effective technique for locating layout geometry configurations sensitive to any of these residual processing effects.

The pattern matcher system architecture was designed as an integrated EDA tool that fits easily into the design flow. It directly reads industry standard CIF and GDSII hierarchical layout data, constructing a compressed hierarchical polygon database in memory for efficient spatial query. The pattern generator is used to directly create a pattern from a set of equations and optical system parameters, which can be archived for later use. The system includes a cross-platform graphical user interface for visualization of the layout, patterns, and match locations found, including screen capture capabilities. The layout visualization portion of the tool is competitive with commercial layout viewers and supports a variety of measurement, query, and image capture features. The pattern matcher also automatically extracts geometry from the resulting match locations for directly simulating aberrations and other optical effects in SPLAT. Since the software architecture is modular, additional components can be added to extend the functionality of the pattern matcher and to allow matching for a diverse set of applications and match requirements.

The physical theory of lens aberrations has been discussed in detail and led to the initial idea of locating areas sensitive to lens aberrations through a pattern matching approach. Lens aberration patterns are generated by first taking the inverse Fourier transform of the aberrated pupil function as defined by a set of Zernike polynomials. A perturbational approach based on the approximation that $e^{jOPD} \approx 1 + jOPD$ is used to estimate the electric field change at a feature edge by convolving the layout with the aberrated point spread function represented by the pattern. This was found to be an accurate approximation of the effects of aberrations on electric field change for small amounts of aberration as verified through SPLAT simulations. Intensity change can be predicted by determining the unaberrated and optical proximity electric field vectors, combining the three electric fields, and calculating the change in intensity, $I = E \cdot E^*$. Line edge shift can also be predicted by dividing the intensity change by the slope at the feature edge, which can either be calculated with a fast aerial image simulation or read from a lookup table of common feature configurations.

The pattern matcher software was built from the ground up but uses concepts and algorithms from other fields such as image processing, computational geometry, computer graphics, and physical CAD. The final implementation was a distinct approach designed to meet the needs of full-chip pattern matching through customized algorithms. The algorithms needed to perform this type of pattern matching differed from image correlation, video compression, and geometric matching methods because of the much larger search space, large groups of identical pixel values, complex number pattern pixel and layer weights, inexact matching, and filtering methods that were used.

A number of important data structures and algorithms were developed to support fast, memory efficient pattern matching on large integrated circuit mask layouts. These fall into the layout preprocessing and pattern pre-integration steps and the actual match factor computation phase. The layout preprocessing involves first reading a layout and building a compressed hierarchical polygon database in memory. This is done to minimize both load time and the size of the database in memory, which is the only limiting factor for the size of a layout that can be processed. The layout is iteratively spatially subdivided into optimal sized polygon bins, and the geometry is spatially sorted for efficient overlap query. As the hierarchy is locally flattened, polygons are split into primitives such as rectangles and triangles, overlaps are removed, Boolean layer operations are computed, and the resulting shapes are merged into a minimum number of primitives. Polygons are split through a recursive algorithm that divides the polygon into smaller polygons with horizontal and vertical cutlines until the sub-polygons become simple rectangles and right triangles. The other geometry operations are accomplished by dividing the working dataset into small regions, checking each shape in a region against all other shapes to compute the resulting set, and then stitching the shapes back together across region boundaries to form maximally large rectangles and triangles. These algorithms were found to be sufficient in processing the polygonal shapes found in most common layout styles, including digital and analog circuits and MEMS devices.

Several pattern matching algorithms have been developed as the system evolved into an efficient large-scale layout processing engine. The initial bitmap algorithm determined the match factor by computing the correlation between the pattern bitmap and the layout bitmap. Due to the extremely long runtimes of this brute-force algorithm, performance improvements such as data compression and edge/corner filtering were developed to reduce both the number of match locations and the number of operations per location. The real speedup came with the idea of pre-integrating the pattern so that only the pixels along feature edges needed to be processed, which led to the edge-intersection algorithm and an order of magnitude improvement in runtime. This 1D pre-integration idea was further extended to 2D, resulting in the rectangle algorithm and a second order of magnitude reduction in runtime. Finally, triangles were added to the rectangle algorithm in order to efficiently represent the 45-degree diagonal edges common to some design technologies. The final filtered rectangle algorithm allowed pattern matching on a full chip to finish in only ten to thirty minutes on a single processor machine, two orders of magnitude faster than OPC. The parallel processor implementation of this algorithm resulted in a near perfect 28X speedup on 32 processors.

There is still significant work that can be done in the area of pattern matching for finding areas subject to adverse processing effects. The pattern matcher accuracy can be verified more rigorously by printing a mask with a stepper or scanner having known lens aberrations and examining the resulting shapes to determine if the locations found in the pattern matcher are in fact the locations of highest line edge shift. Additional work is required to actually verify the pattern matcher theory for physical applications such as CMP, reflective notching, and laser-assisted thermal processing, which will require printing test wafers involving multiple masks and multiple process steps. Several architectural and algorithmic extensions are also possible. The pattern matcher could be integrated with the fast image intensity simulators of OPC algorithms to calculate intensity change and line edge shift from the electric field change predicted by the match factor. Triangle layer operations and non-45 degree, arbitrary angle pattern pre-integration and matching must be implemented in order to process layouts such as MEMS devices and analog circuits that include polygons with odd angles. Finally, it may be possible to combine the current rectangle/triangle pattern matching algorithm with other FFT-based, learning, statistical, or rejection algorithms for a hybrid approach that outperforms all individual methods.

The pattern matcher software is a novel and intriguing approach to improving the manufacturability of integrated circuit devices early in the design stage. This system is a fast, inexpensive, proven alternative to the iterative mask design process that can be applied to many areas of DFM and perhaps even other complex optical systems outside of the semiconductor area. This software system should make a useful addition to a mask designer's CAD toolset.

References

[1] B. Grenon, direct communication.

[2] D. G. Flagello, R. J. Socha, X. Shi, J. B. van Schoot, J. Baselmans, M. A. van de Kerkhof, W. de Boeij, A. Engelen, R. Carpaij, M. H. P. Moers, M. Mulder, M. Schriever, M. Maul, H. Haidner, M. Goeppert, U. Wegmann, P. Graeupner, "Optimizing and Enhancing Optical Systems to Meet the Low k_1 Challenge", *Proc. SPIE*, Vol. 5040, Feb. 2003.

[3] L. W. Liebmann, S. M. Mansfield, A. K. Wong, M. A. Lavin, W. C. Leipoid and T. G. Dunham, "TCAD Development for Lithography Resolution Enhancement", *IBM Journal of Research and Development*, 45/5, pp. 651-665, 2001.

[4] W. Grobman, M. Thompson, R. Wang, C. Yuan, R. Tian, and E. Demircan," Enhancement Technology: Implications and Challenges for Physical Design", *Proc. Design Automation Conf.*, Las Vegas, NV, 2001, pp. 73-78.

[5] J. Stirniman and M. Rieger, "Fast Proximity Correction with Zone Sampling," in Optical/Laser Microlithography VII, ed. T. Brunner, *Proc. SPIE*, Vol. 2197, pp. 294-301, 1994.

[6] N. Cobb, A. Zakhor, "Experimental Results on Optical Proximity Correction with Variable Threshold Resist Model," *Proc. SPIE*, Vol. 3051, pp. 458-468, 1997.

[7] Y.C. Pati, A.A. Ghazanfarian, R.F. Pease, "Exploiting Structure for Fast Aerial Image Computation for Integrated Circuit Patterns," IEEE Trans. Semiconductor Manufacturing, Vol. 10(1), pp. 62-74, February 1997. Vol. 4692-54, 2002.

[8] S. Smith, A.J. Walton and M. Fallon, "Investigation of Optical Proximity Correction (OPC) and Non-Uniformities on the Performance of Resistivity and Linewidth Measurements", *Proceedings of IEEE International Conference on Microelectronic Test Structures (ICTMS99)*, pp. 161-166, March 1999.

[9] F.M. Schellenberg, L. Capodieci and B. Socha, "Adoption of OPC and the Impact on Design and Layout", *Proc. ACM/IEEE 38th Design Automation Conference*. pp. 89-92, June 2001.

[10] N. Cobb, "Fast Optical and Process Proximity Correction Algorithms for Integrated Circuit Manufacturing," PhD Dissertation, University of California, Berkeley, pp. 38-45, 1998.

[11] V. N. Mahajan, *Aberration Theory Made Simple*, SPIE Optical Engineering Press, p. 74, 1991.

[12] T. A. Brunner, "Impact of Lens Aberrations on Optical Lithography," *IBM J. Res. Develop.* 41, Jan./Mar. 1997.

[13] J. Kirk, "Review of Photoresist-Based Lens Evaluation Methods," *Proc. SPIE,* Vol. 4000, pp. 2, 2000.

[14] T. A. Brunner, A. L. Martin, R. M. Marino, C. P. Ausschnitt, T. H. Newman, M. S. Hibbs, "Quantitative Stepper Metrology Using the Focus Monitor Test Mask," *Proc. SPIE,* Vol. 2197, pp. 541-549, 1994.

[15] P. Dirksen, J. J. M. Braat, A. Janssen, A. Leeuwestein, C. A. H. Juffermans, "Experimental Determination of Lens Aberrations from the Intensity Point-Spread Function", *Proc. SPIE*, Vol. 5040, Feb. 2003.

[16] P. Dirksen, C. A. Juffermans, A. Engelen, P. De Bisschop, H. Muellerke, "Impact of High Order Aberration on the Performance of the Aberration Monitor", *Proc. SPIE*, Vol.4000, p.9 July 2000.

[17] P. Dirksen, C. A. Juffermans, R. J. Pellens, M. Maenhoudt, P. De Bisschop, "Novel Aberration Monitor for Optical Lithography", *Optical Microlithography XII, Proc. SPIE*, Vol. 3679, March 1999.

[18] A. Imai, K. Hayano, H. Fukuda, N. Asai, N. Hasegawa, "Lens Aberration Measurement Technique Using Attenuated Phase-Shifting Mask", *Optical Microlithography XIII, Proc. SPIE*, Vol. 4000, July 2000. Note: The title of this reference has been misspelled as "Lens Aberration Measurement Technique Using Attentuated Phase-Shifting Mask" in some locations.

[19] J. W. Goodman, *Introduction to Fourier Optics, 2nd Edition*, McGraw Hill, p. 155, 1996.

[20] M. Born and E. Wolf, *Principles of Optics, 7th Edition*, Section 9.4, Cambridge University Press, 1999.

[21] F. Gennari, A. Neureuther, "Aberrations are a Big Part of OPC for Phase-Shifting Masks," *Proc. SPIE*, Vol. 4562, 10/01.

[22] A.R. Neureuther, "Modeling Phase Shifting Masks," *Proc. SPIE*, Vol.1496, pp. 80- 88, 1990.

[23] F. Gennari, "Pattern Matcher for Locating Areas in Phase-Shift Masks Sensitive to

Aberrations", M.S. Project Report, University of California Berkeley, Dec. 2001.

[24] H. Fukuda, K. Hayano and S. Shirai, "Determination of High-Order Lens Aberrations Using Phase/Amplitude Linear Algebra," *J. Vac Sci. Technol. B.* 17(6), pp. 3318-3321, Nov/Dec 1999.

[25] Mark Terry of Texas Instruments, Private Communication, March 2001.

[26] A. R. Neureuther, S. Hotta, K. Adam, "Modeling Defect-Feature Interactions in the Presence of Aberrations," *Proc. SPIE*, Vol. 4186, pp. 405-414, 2001.

[27] A. R. Neureuther, P. Flanner III and S. Shen, "Coherence of Defect Interactions with Features in Optical Imaging," *J. Vac. Sci. Technol. B*, pp. 308-312, Jan/Feb. 1987.

[28] A. Wong, *Resolution Enhancement Techniques in Optical Lithography*, SPIE Press, Bellingham, WA, 2001.

[29] G. Robins and A. Neureuther, "Illumination, Resists, Mask, and Tool Effects on Pattern and Probe-Based Aberration Monitors," *Proc. SPIE*, Vol. 4691, April 2002.

[30] G. Robins, A.R. Neureuther, K. Adam, "Measuring Optical Image Aberrations with Pattern and Probe Based Targets," *J. Vac. Sci. Technol. B*, Vol. 20, No. 1, pp. 338-343, Feb. 2002.

[31] C. M. Garza, W. Conley, B. J. Roman, M. Schippers, J. Foster, J. Baselmans, K. D. Cummings, D. G. Flagello, "Ring Test Aberration Determination and Device Lithography correlation," pp.36-44, *Proc. SPIE*, Vol. 4345, pp.36-44, 2000.

[32] K. H. Toh and A. R. Neureuther, "Identifying and Monitoring Effects of Lens Aberrations in Projection Printing," *Proc SPIE*, Vol. 772, p. 202, 1987.

[33] S. M. Rubin, *Computer Aids for VLSI Design, 2nd Edition*, Appendix C, 1994. http://www.rulabinsky.com/cavd/

[34] P. Rai-Choudhury, *SPIE Handbook of Microlithography, Micromachining and Microfabrication*, Section 2.9, 1987. http://www.nnf.cornell.edu/SPIEBook/spie9.htm

[35] S. M. Rubin, *Computer Aids for VLSI Design, 2nd Edition*, Appendix B, 1994. http://www.rulabinsky.com/cavd/

[36] V. Dai and A. Zakhor, "Lossless Compression Techniques for Maskless Lithography Data," *Emerging Lithographic Technologies VI, Proceedings of SPIE*, Vol. 4688, 2002.

[37] A.J. Reich, R.E. Boone, W.D. Grobman, C. Browning, "GDSII Considered Harmful", *Proc. SPIE 4562*, Oct. 2001.

[38] M. Niewczas, W. Maly and A. Strojwas, "A Pattern Matching Algorithm for Verification and Analysis of Very Large IC Layouts", *Proc. of International Symposium on Physical Design*, Monterey, CA, April 1998, pp. 129-134.

[39] K. Lee and A.R. Neureuther, "SIMPL-2 (SIMulated Profiles from the Layout - Version 2)," *1985 Symposium on VLSI Technology*, Kobe, Japan, Digest of Technical Papers, pp. 64-65, May 1985.

[40] C. Hsu, R. Yang, J. Cheng, P. Chien, V. Wen, A. R. Neureuther, "Lithography Analysis Using Virtual Access (LAVA) Web Recourse, "*Proc. SPIE* Vol. 3334, pp. 197-201, 1998.

[41] F. Gennari, S. Madahar, and A. Neureuther, "Web Tool for Worst Case Assessment of Aberration Effects in Printing a Layout," *Proc. SPIE*, Vol. 5040, 2/27/03.

[42] F. Gennari, A. Neureuther, "A Pattern Matching System for Linking TCAD and EDA", *ISQED* 3/04.

[43] F. Gennari, "Linking TCAD and EDA Through Pattern Matching," *SRC Techcon* 2003, 8/26/03.

[44] Dinesh Nair, Ram Rajagopal and Lothar Wenzel, "Pattern Matching Based on a Generalized Fourier Transform", *Proc. SPIE*, Vol. 4116, 2000.

[45] Ram Rajagopal, "Pattern Matching Based on a Generalized Transform", Report, National Instruments, 2000.

[46] Swami Manickam, Scott D. Roth, Thomas Bushman, "Intelligent and Optimal Normalized Correlation for High- Speed Pattern Matching", Product Description, Datacube, Inc.

[47] Yakov Hel-Or and Hagit Hel-Or, "Real Time Pattern Matching Using Projection Kernels," *ICCV03* (1486-1493).

[48] M. Elad, Y. Hel-Or, and R. Keshet. "Pattern Detection Using Maximal Rejection Classifier." *IWVF4 - 4th International Workshop on Visual Form*, pages 28-30, Capri, Italy, May 2000.

[49] M. Atallah, Y. Genin, W. Szpankowski, "Pattern Matching Image Compression: Algorithmic and Empirical Results", *Proc. International Conference on Image Processing*, Vol. II. 349-352, Lausanne, 1996.

[50] D. Arnaud and W. Szpankowski, "Pattern Matching Image Compression with Prediction Loop: Preliminary Experimental Results", Purdue University, CSD-TR-96-069, 1996.

[51] M. Keil, Polygon Decomposition, in: J.-R. Sack, J. Urrutia (Eds.), Handbook of Computational Geometry, Elsevier Science/North-Holland, Amsterdam, 1999.

[52] D. Hearn, M. P. Baker, *Computer Graphics: C Version, 2nd Edition*, Prentice Hall, Upper Saddle River, NJ, Chapter 3, 1997..

[53] M. Bern, S. Mitchell, and J. Ruppert, "Linear Size Nonobtuse Triangulation of Polygons", *Discrete & Computational Geometry*,14:411-428, 1995.

[54] J. Shewchuk, "Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator", *First Workshop on Applied Computational Geometry* (Philadelphia, Pennsylvania), pp. 124-133, Association for Computing Machinery, May 1996.

[55] J. Neider, T. Davis, *OpenGL Programming Guide, 2nd Edition*, Addison-Wesley Publishing Company, Chapter 12, 1997.

[56] U. Lauther, "An O(N log N) Algorithm for Boolean Mask Operations", *18th Design Automation Conference*, pp. 555-560, 1981.

[57] A. Yarkhan and L. Manne, *Beginners Guide to MPI Message Passing Interface*, Manual, 1997.

[58] B. Nichols, D. Buttlar, J. P. Farrell, *Pthreads Programming: A POSIX Standard for Better Multiprocessing*, O'Reilly, 1996.

[59] F. Gennari, G. Robins, and A. Neureuther, "Validation of the Aberration Pattern-Matching OPC Process," *Proc. SPIE,* Vol. 4692B, 3/02.

[60] A. R. Neureuther, F. Gennari, "No-Fault Assurance: Linking Fast Process CAD and EDA," *Proc. SPIE*, Vol. 4889, 10/02.

[61] L. Liebmann, B. Grenon, M. Lavin, S. Schomody, T. Zell, "Optical Proximity Correction, a First Look at Manufacturability," Proc. SPIE Vol. 2322, 1994.

[62] Y. Granik, "Correction for Etch Proximity: New Models and Applications," *Proc SPIE* Vol. 4246, pp. 98-112, 2001.

[63] C. Progler, A. Wong, "Zernike Coefficients: Are They Really Enough?", *Proc SPIE* Vol. 4000, pp 40-52, 2000.

[64] K. Lai, C. Wu, C. Progler, "Scattered Light: The Increasing Problem for 193nm Exposure Tools and Beyond", *Proc. SPIE* Vol. 4346, 2001.

[65] S. Hafeman, F. Gennari, and A. Neureuther, "Fast Algorithm for Extraction of Worst-Case Image Degradation Due to Flare", *Proc. SPIE,* Vol. 5377, 2/04.

[66] J. W. Goodman, *Statistical Optics*, Wiley & Sons, New York, Chapter 8, 1985.

[67] J. Kirk, "Scattered Light in Photolithographic Lenses", *Proc. SPIE* Vol. 2197, pp 566-572, 1994.

[68] Neil Callan and Ebo Croffie of LSI Logic, direct communication.

[69] V. Mastromarco, K.H. Toh and A.R. Neureuther, "Printability of Defects in Optical Lithography: Polarity and Critical Location Effects," *J. Vac. Sci. Technol. B*. pp. 224-229, Jan./Feb. 1988.

[70] R. Socha and A. Neureuther and R. Singh, "Printability of Phase-Shift Defects using a Perturbational Model," in BACUS *Proc. 13 Annu. Symp. Photomask Technology Management*, pp. 277-287, 1993.

[71] D. J. Bald, S. Munir, B. Lieberman, W. H. Howard, C. A. Mack, "PRIMADONNA: A System for Automated Defect Disposition of Production Masks Using Wafer Lithography Simulation", *Proc. SPIE*, Vol. 4889, Dec. 2002.

[72] M. S. Yeung and A. R. Neureuther, "Three-Dimensional Reflective-Notching Simulation Using Multipole Accelerated Physical-Optics Approximation", *Proc. SPIE*, Vol. 2440, 395-409 (1995).

[73] J. Word., D. Chou, Y. Gu, J. Sturtevant, "Reduction of Reflective Notching Through Illumination Optimization," *Proc. SPIE*, Vol. 4691, pp. 990-998, 2002.

[74] D. M. Camm, J. C. Gelpey, T. Thrum, G. C. Stuart, J. K. Elliott, "Engineering Ultra-Shallow Junctions Using fRTP", Vortek Industries Limited.

[75] A. Shima et al., "Ultrashallow Junction Formation by Non-melt Laser Spike Annealing for 50nm Gate CMOS," *Proceedings of 2004 Symposia on VLSI Technology and Circuits*, Honolulu, HI, June 15-19, 2004.

[76] S. K. H. Fung et al., "65nm CMOS High Speed, General Purpose and Low Power Transistor Technology for High Volume Foundry Application," *Proceedings of 2004 Symposia on VLSI Technology and Circuits*, Honolulu, HI, June 15-19, 2004.

[77] I. Rangelow, "Critical Tasks in High Aspect Ratio Silicon Dry Etching for Microelectromechanical Systems", *Journal of Vacuum Science Technology*, Volume 21, Issue 4, pp. 1550-1562 (2003).

[78] G. Fu, A. Chandra, "An Analytical Dishing and Step Height Reduction Model for Chemical Mechanical Planarization (CMP)", *IEEE Transactions on Semiconductor Manufacturing*, Vol. 16, No. 3, pp. 477-485, Aug. 2003.

[79] G. Nanz and L. E. Camilletti, "Modeling of Chemical-Mechanical Polishing: A Review", *IEEE Trans. on Semiconductor Manufacturing* 8(4), pp. 382-389, 1995.

[80] A. B. Kahng, G. Robins, A. Singh, H. Wang, and A. Zelikovsky, "Filling and Slotting: Analysis and Algorithms", *Proceedings of the ACM International Symposium on Physical Design*, pp. 95-102, April 1998.

[81] G. McIntyre and A. Neureuther, "Interferometric-Probe Monitors for Self-Diagnostics of Phase-Shifting Mask Performance." *Proc. SPIE,* Vol. 5130, 2003.

[82] G. Cellere, L. Larcher, M. G. Valentini and A. Paccagnella, "Plasma-Induced Micro Breakdown in Small-Area MOSFETs" *IEEE Transactions on Electron Devices*, Vol. 49, No. 10, Oct. 2002.